Document Title	Guide to BSW Distribution
Document Owner	AUTOSAR
Document Responsibility	AUTOSAR
Document Identification No	631

Document Status	Final
Part of AUTOSAR Standard	Classic Platform
Part of Standard Release	4.3.1

Document Change History

Date	Release	Changed by	Change Description
2017-12-08	4.3.1	AUTOSAR Release Management	• Editorial changes
2016-11-30	4.3.0	AUTOSAR Release Management	• Editorial changes
2014-10-31	4.2.1	AUTOSAR Release Management	• Incorporation of concept “Mechanisms and constraints to protect ASIL BSW against QM BSW” • Minor clarifications
2014-03-31	4.1.3	AUTOSAR Release Management	• Clarified terms
2013-03-15	4.1.1	AUTOSAR Administration	• Initial release

Disclaimer

Table of Contents

1 Introduction

This document is a general introduction to the distribution of BSW in AUTOSAR systems. It consists of two parts, one focusing on the distribution of BSW in case of multi-core and the other focusing on distribution in case of safety.

Chapter 2 guides to the development and configuration of AUTOSAR-compliant software for multi-core systems. As of release 4.1, it addresses the allocation of AUTOSAR BSW modules [1] to partitions on multi-core systems and their interaction only. The allocation of BSW modules to different BSW partitions allows for both enhanced functional safety and increased performance.

In chapter 3 the BSW distribution in safety cases is described. As of release 4.2 AUTOSAR allows to map BSW modules into different partitions and to protect those partitions against each other.

Chapter 4 gives an outlook of possible future extensions in the area of BSW distribution.

A glossary of technical terms and a list of references to external information are provided in chapters 5 and 6.

2 BSW Distribution in Multi-Core Systems

2.1 Overview

This chapter contains a description of the supported scenarios for distributed execution of BSW modules on several partitions and cores and a number of use cases in which a distribution of the BSW can enhance performance. It also introduces basic synchronization concepts applicable to distributed BSW execution, and an introduction to inter-partition communication.

2.1.1 Supported Scenarios

It is possible to assign functional clusters of BSW modules (“"BSW Functional cluster"), which are used by applications to access buses, non-volatile memory, I/O channels, and watchdogs, to different BSW partitions for safety or performance reasons. The clustering of BSW modules is currently not standardized. Parallel usage of the same type of functional clusters in different partitions ("duplication") is not generally supported, but it is possible by using a master satellite approach. Functional clusters to partitions may be assigned such that

- a BSW functional cluster is only available in one partition
- a BSW functional cluster is available on all partitions with all interfaces
- a BSW functional cluster is distributed over multiple partitions, possibly with partition specific subsets of functionality, to allow a high grade of concurrency.

In either of these scenarios, the following restrictions apply:

- There is currently at most one QM BSW partition per core.

With the aforementioned restrictions, AUTOSAR supports the scenarios listed above. In doing so, it addresses the following essential features:

- All code for communication between BSW partitions can be generated for automatic adaptation to different system configurations. The cross partition communication mechanism can be generated with focus on efficiency, or, in future releases to help to provide freedom of interference.
- If access to system services (which are not part of a BSW functional cluster) is required, efficient access from each BSW partition that needs the system service is supported.
- Efficient access to HW abstraction and drivers is supported in each BSW partition, if required.

In all scenarios, the communication between different module entities remains unchanged (in comparison to BSW running in a single partition).

2.1.2 Performance Use Cases and Hardware Assigned to Different Cores

The following use cases are examples for how system performance can be improved by allocation of the BSW to multiple partitions and cores, and how systems where the access to the peripheral hardware is assigned to multiple cores benefit from the allocation of the BSW to multiple partitions and cores.

- To increase system performance and to reduce resource consumption in systems that are distributed over several cores, it may be necessary to allocate functional clusters of BSW modules to different cores, e.g.

communication modules on BSW partition "A" and I/O modules on BSW partition "B", depending on hardware architecture, load balancing and on distribution of SW-Cs. In particular, if HW resources are accessed exclusively by one core in a Multi-Core system, the performance is increased by locating the corresponding BSW users, services and drivers on that core.

- Signal gateway functionality is implemented by allocating a FlexRay cluster on one core and a CAN cluster on a different core. The two COM modules need to be synchronized in this case, and there must be some direct cross core communication between the two COM instances. One of the COM modules might be the master COM that coordinates the satellite COM on the other core.
- Two communication clusters are located on different cores, one accessing a CAN bus and the other one controlling a FlexRay bus. In case the application SW located above one of the communication clusters on the same core needs to send on both buses, the core local COM modules can directly communicate with their counterparts on the other core, to efficiently send the signal over either CAN or FlexRay. For received messages, COM has no information about receivers above the RTE. Therefore, COM has to forward the signals on the receiving side to the RTE, and the RTE is responsible for communication.

2.1.3 Technical Overview

Below is a short summary of the technical solution as described in the following sections:

- Define clusters of BSW modules that contain preferably all three layers of a stack, or, if needed, a subset of modules of a stack (e.g. communication, memory, I/O stack).
- Module entities can be split into a master and satellites, which are assigned to different BSW partitions. Masters and satellites can use non-standardized AUTOSAR interfaces, for internal cross partition communication. The master/satellite approach is mainly used by distributed system service modules and for communication between BSW clusters of the same type.

The proposed solution meets the demands on performance and safety while minimizing the impact on already standardized BSW module interfaces (RS_BRF_00206, RS_BRF_01160). Most changes are hidden within modules (e.g. by providing master/satellite implementations) without affecting other modules. Interfaces between different modules do not change.

2.1.3.1 BSW Functional Clusters

BSW functional clusters are groups of functionally coherent BSW modules. Each functional cluster includes a set of BSW modules. It is possible to have several BSW functional clusters of the same type (e.g. several I/O clusters in different BSW partitions), each using a different set of modules (e.g. IOHWA + ADC in one partition and IOHWA + ADC + DIO in the second partition).

The following types of clusters might be standardized in a later release:

- Communication cluster
- Memory cluster
- I/O cluster

- Watchdog cluster

The allocation of BSW functional clusters to BSW partitions is determined by the usage of BSW modules by the application software. Functional clusters can be allocated to different BSW partitions, and functional clusters of the same type can be available in several BSW partitions. Different functional clusters can be allocated to the same or to different BSW partitions.

The same functional cluster can only exist at most once per BSW partition.

BSW functional clusters are used by applications or other BSW modules to access buses, memory, I/O channels and watchdogs, and they are usually required in one or few BSW partitions only.

The introduction of BSW functional clusters does not change the existing AUTOSAR R4.0 interfaces between the BSW and the RTE, which are mainly used to implement AUTOSAR services, i.e. to communicate with the application layer. It may however change the availability of standardized AUTOSAR interfaces on different partitions.

The internal structure of a BSW functional cluster, including its internal communication between BSW modules, and the communication with system services that the BSW functional cluster uses is not necessarily affected by the parallelization of the BSW, and it does not need to change. It may however be adapted, for example in order to fulfill special demands on concurrency like the support of different entities of the same module running in different partitions.

The communication and synchronization between modules in BSW functional clusters of the same type (e.g. in two communication clusters to support a gateway functionality) is not standardized. It will be implemented by communication between entities (e.g. by a master and satellites) of specific modules, which can use non-standardized interfaces for communication across BSW partition boundaries, see Figure 1.

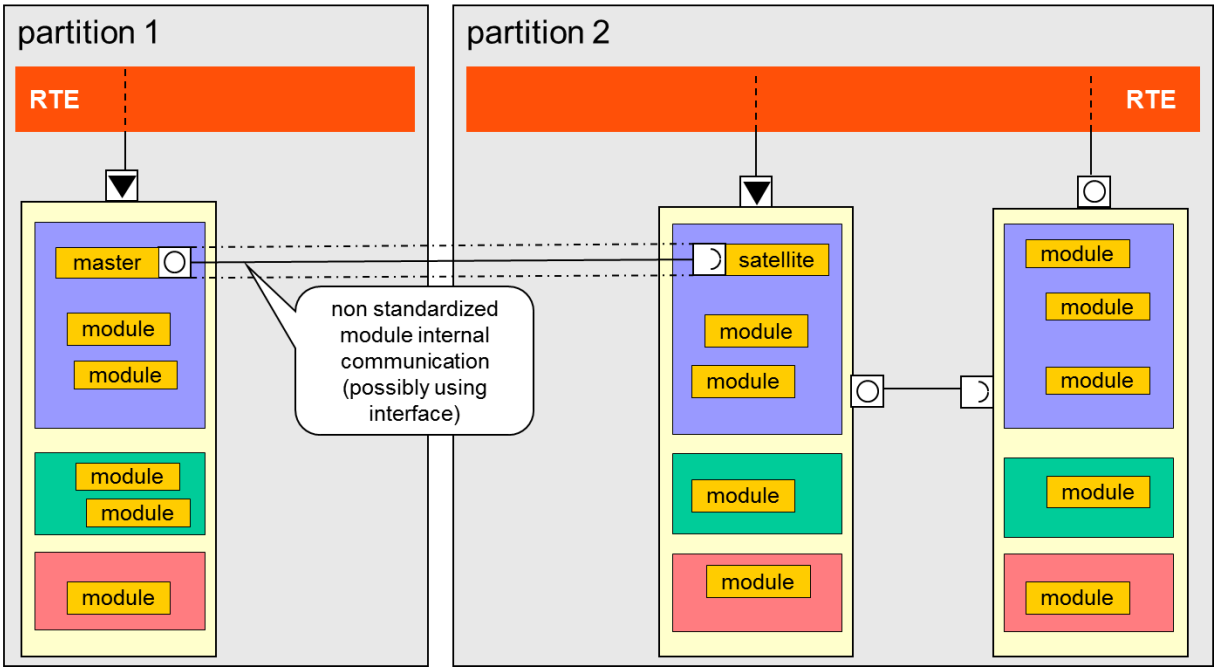

Figure 1: Functional clusters of the same type

Modules that do not belong to BSW functional clusters (some MCAL modules, system services) will always be accessed within the same BSW partition where the BSW functional cluster is located. As the interfaces do not change, these modules must be locally available in each BSW partition, if needed.

2.1.3.2 Inter-BSW-partition communication

Function calls to tasks that are supposed to be executed in a different BSW partition/on a different core cannot be implemented as simple C calls to this function, because these calls would be handled on the local BSW partition.

The BSW Scheduler (SchM) therefore provides functions to invoke masters or satellites of the same module on different BSW partitions using either client-server or sender-receiver communication. Details on this API of the SchM are explained in Section 2.2.3.

2.1.3.3 Determining the Partition for Service Execution

The actual BSW partition for the handling of an RTE event is determined by its task mapping. Basically, if an event is mapped to a task, it is executed within the partition assigned to this task. If an event is not mapped to a task, it is executed within the same partition as the task that caused the event. Details on the task mapping are described in Section 2.4.1 of this document.

Calls from BSW entities to other BSW entities are not mapped to a partition. They are executed wherever they are called. Therefore, several calls to a BSW function may be processed in parallel on different partitions and cores. Consequently such functions must be designed and implemented carefully w.r.t. parallel execution in different partitions; if necessary, they shall be reentrant or concurrency safe.

2.1.3.4 BSW partitions

Only partitions that have the configuration parameter *EcucPartitionBswModuleExecution* set to true can execute BSW modules. Such partitions are called BSW partitions. BSW partitions may additionally contain application software components above the RTE.

2.2 Parallel Execution of BSW modules

This is the chapter for developers of BSW modules.

2.2.1 Core-Dependent Branching

Because entities of the same module share the same implementation, even if they are running on different cores, different behavior cannot be realized by different code. Instead, the specific behavior shall be determined by runtime information. It is possible for example to use the core id for this, i.e. branch the control flow depending on the return value of the OS APIs `GetCoreID()`, or also `GetApplicationID()`.

2.2.2 Master/Satellite-approach

Modules that need to be accessed in different BSW partitions can be implemented using the master/satellite pattern.

The distribution of work between master and satellite is implementation specific. One extreme is that the satellite only provides the interfaces to the other modules in the same BSW partition, and that it routes all requests to the master and answers back to the other modules. At the other extreme, the satellite can provide the full functionality locally (e.g. local mode management for a complete application which runs in the same BSW partition) and only synchronizes its internal states with the master, if necessary. There might even be several masters for different functionality, e.g. two PduR masters for a distributed PduR gateway.

The master coordinates requests from the satellites and can filter or monitor incoming satellite requests. The master and one or several satellites are treated like being one module entity in some respect:

- Master and satellites are always vendor specific solutions, coming from the same vendor.
- The interfaces of master and satellite to other module entities in general are the same as specified in AUTOSAR R4.0 for traditional modules. Master and satellite should provide the same APIs. This means that when migrating to partitioned systems, existing module entities can be replaced by a master and one or several satellites, in most cases without changing other modules. Exceptions might be module internal adaptations to additional delays which are caused by inter-partition communication.
- Master and satellites have the same entry points in each BSW partition (i.e. they start executing the same functions from shared memory) and internally branch (e.g. by using the "GetApplicationID ()" API) to master or satellite specific code according to the OS-Application (partition) they run in. Depending on the build strategy, other implementations might be possible in multi-Core systems if each core can execute its own code. Also, satellites might share the same code without further branching.

- The communication between master and satellites is not standardized. It is considered to be module-internal and is not visible to other modules.
- The communication between master and satellite can be initiated in either direction (i.e. by both the master and the satellites), as well as from one satellite to another one.
- All interfaces between masters and satellites are only allowed to be connected within the same distributed module.
- The communication between master and satellites can be implemented within one BswModuleEntity, or between different BswModuleEntities that belong to the same BSW module.
- Depending on the application, usage of master/satellite may be appropriate or not. For example, it may be more efficient to use separate, partition specific watchdog clusters, which work independently from each other, rather than using the Watchdog Manager in a master/satellite approach.
- The master is the part of a distributed BSW module that coordinates requests by satellites and can filter or monitor incoming satellite requests. This may result in additional fault detection or fault mitigation mechanisms. Generally, all errors caused by distributed execution of a module should be handled module internally.

The master/satellite implementation is the standard solution for system services in partitioned systems.

Specific drivers also might have to provide local satellites, if the hardware can only be accessed from a different core. The standard solution, if possible, is to execute the same multi-core reentrant function in each partition and to separate the data to work on into disjoint sets, one for each partition. For example, the COM module may work on all IPDUs assigned to the bus that the BSW functional cluster of this module belongs to. Concurrent access to the same hardware or shared data needs to be protected, e.g. by ExclusiveAreas in this case.

In specific cases, modules within BSW functional clusters also need to be implemented as master/satellite, if the BSW functional clusters are duplicated and the entities in different BSW partitions need to be synchronized or need to exchange data. This might apply to the Watchdog Manager, the NVRAM manager, and to network and state managers in duplicated communication clusters. COM modules also might need to have a master and a satellite to implement cross partition gateway functionality.

2.2.3 Using the BSW Scheduler for Inter-Partition-Communication

The BSW Scheduler (SchM) provides a number of functions to support communication between BSW module entities that are executed in parallel. More precisely, it provides the following methods to handle synchronous and asynchronous calls (including callbacks) as well as sender-receiver communication.

The functionality is generally similar to that of function calls between SWCs and the BSW. However, because the RTE may not be available at certain points of time (especially during startup of an ECU), this functionality must be available within the BSW itself.

- `Std_ReturnType SchM_Call_<bsnp>[_<vi>_<ai>]_<name>(`
  `[OUT <typeOfReturnValue> returnValue]`
  `[IN|IN/OUT\|OUT]<data_1> ... [IN|IN/OUT|OUT] <data_n>)`
  or
  `Std_ReturnType SchM_Call_<bsnp>[_<vi>_<ai>]_<name>(`
  `[IN|IN/OUT\|OUT]<data_1> ... [IN|IN/OUT|OUT] <data_n>)`

  Invoke a client-server-operation, possibly crossing partition boundaries. The actual parameters `data_1 ... data_n` are information that is passed [IN] and/or re-passed [IN/OUT | OUT] to/from the called service.

  The presence of the parameter `returnValue` and its type `<typeOfReturnValue>` depend on the called service. For synchronous calls, the parameter is present and `<typeOfReturnValue>` is the type returned by the called service. For asynchronous client-server-operations and operations with return type `void`, the parameter is omitted.

- `Std_ReturnType SchM_Result_<bsnp>[_<vi>_<ai>]_<name>(`
  `[IN|IN/OUT|OUT]<data_1> ... [IN|IN/OUT|OUT] <data_n>)`

  Callback from an asynchronous client-server-operation, possibly crossing partition boundaries.

  The receiver of a callback is determined by the AsynchronousServerCallResultPoint of this callback. The AsynchronousServerCallResultPoint refers to the originating AsynchronousServerCallPoint, which in turn “knows” the calling module entity.

- `Std_ReturnType SchM_Send_<bsnp>[_<vi>_<ai>]_<name>(IN <data>)`

  Write data to a sender-receiver link between BSW modules, possibly crossing partition boundaries.

- `Std_ReturnType SchM_Receive_<bsnp>[_<vi>_<ai>]_<name>(OUT <data>)`

  Read data from a sender-receiver link between BSW modules, possibly crossing partition boundaries.

2.2.4 Using Shared Buffers (in systems without memory protection)

In systems without memory protection between the BSW partitions, system services and all BswCalledEntities can be called directly in every partition, including the complete call tree. This requires a reentrant, concurrency safe implementation.

The services and other called entities might work on module internal data, which is shared between different entities of the same module. All access to such data must be protected by ExclusiveAreas. Appropriateness of concrete protection mechanisms depends on the possible kinds of access. For example, concurrent writing generally

needs to be prohibited, whereas concurrent reading may be acceptable, as long as only one partition writes at the same time.

BswSchedulableEntities are located on one core only and process the data periodically or event driven.

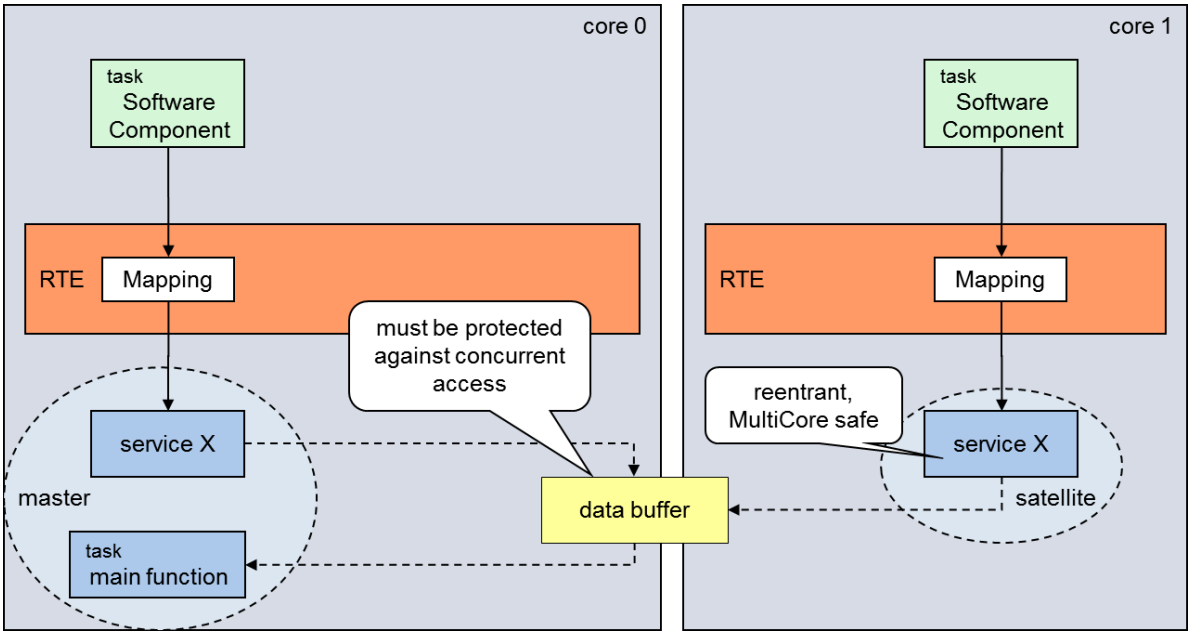

Figure 2: Invocation of same service on different cores

Figure 2 shows the example of a service "X", where the same API and the same code is called directly by the RTE on different cores. This is the default, if the services (respectively the OperationInvokedEvents) are not mapped to a task.

The code must be reentrant and concurrency safe, which means that all access to data must be protected against concurrent access by the same or by a different entity of the same module.

In this example, the same service "X" (BswCalledEntity) writes into a module internal data buffer accessible from core 0 and from core 1. A "main function" (BswSchedulableEntity), which is mapped to a task, reads the data from the buffer for further processing. In order to prevent read/write-conflicts, this “main function” must be protected from reading the buffer while it is written.

This can be considered a special case of the generic master/satellite approach for systems without memory protection between the BSW partitions.

The advantage of this approach is that the original, unchanged modules can be used, as long as they are implemented concurrency safe, which is usually the case for single core already, if different entities of the same module work on the same data, as shown in the example for core 0. Compared to the AUTOSAR R4.0 solution, where all service calls have to be routed to the master core, the performance can be improved considerably without much effort (assuming there is no need to do cross-core communication later).

The following must be considered for a concurrency safe, reentrant implementation:

- Access to all shared resources, e.g. buffers, is protected by ExclusiveAreas.

- Call trees can be made multi-core safe, if either called entities are safe, or calls are protected by ExclusiveAreas (if lock times stay within a specified limit).

BswCalledEntities that are available to CDDs can also be called directly by the CDD. The same rules apply as in R4.0.

The SchM must support cross core ExclusiveAreas, implemented by protected Spinlocks. A protected spinlock is an exclusive area that has "OS_SPINLOCK" as its value of "RteExclusiveAreaImplMechanism". This kind of exclusive areas is available for controlled access by BSW modules only. Protected spinlocks are handled by the Basic Software Scheduler.

2.2.5 Accessing Hardware/Drivers

BswModuleEntities of the MCAL (drivers) are accessed within the BSW partition where the caller is located.

If the same driver is required in different BSW partitions, different types of implementations are possible:

- The same reentrant code can be executed in each BSW partition. The driver can be accessed with the same API in each BSW partition. The code does not need to be partition-aware. This is the default solution, which is safe as long as the hardware objects are exclusively assigned to a single partition.
- The driver is of master/satellite type with internal branching, depending on the BSW partition it is running in. The masters may call other entities in the same BSW partition and access the hardware. The satellites may forward the request to the master in a different BSW partition without accessing any hardware directly.

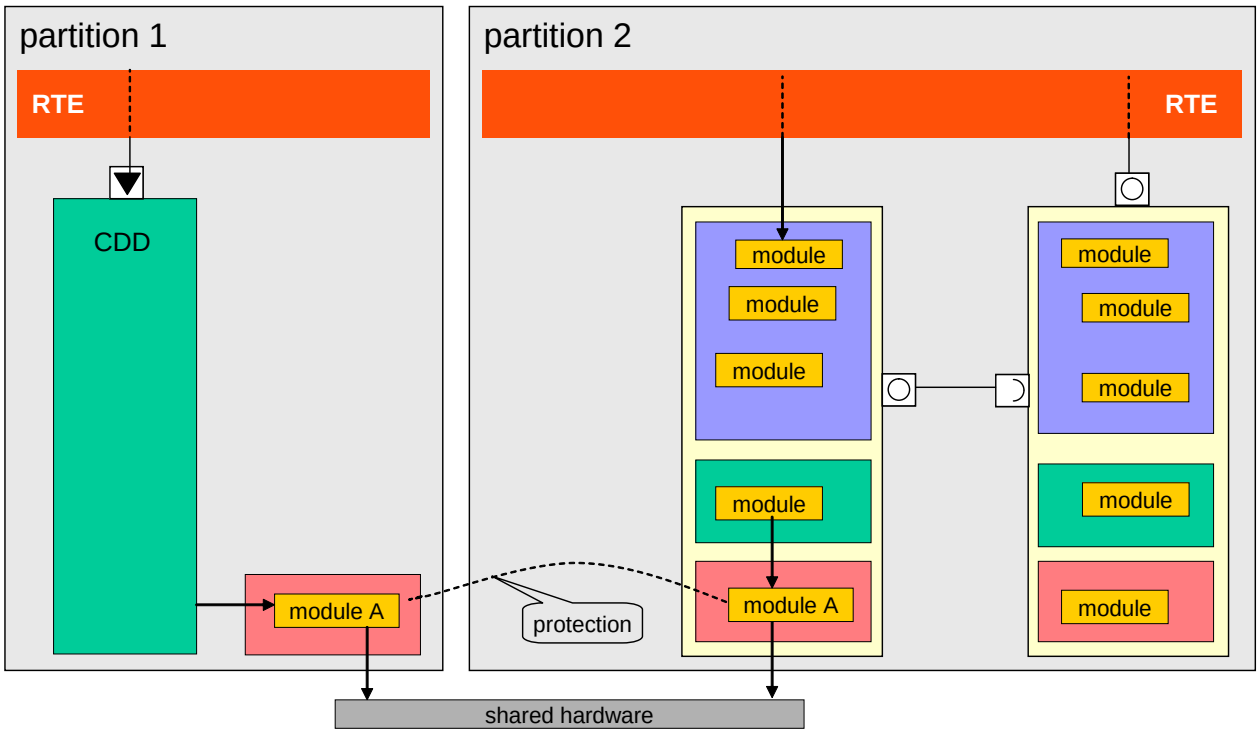

Figure 3: Protected Access to Shared Hardware

In general, the same hardware should only be accessed from one BSW partition. However, if concurrent access to the same hardware from different partitions is unavoidable, this needs to be protected within the MCAL module as shown in Figure 3, e.g. by using ExclusiveAreas. This is particularly important if drivers on different cores access the same hardware.

2.2.6 Concurrency safe implementation of modules

Concurrency safety of BSW modules respectively the functions implemented by these modules may be achieved by different mechanisms.

Generally, the following levels of reentrancy can be distinguished according to (TPS_BSWMDT_04103). The concrete level of a BswModuleEntity is defined in the optional attribute “reentrancyLevel”.

- **Multi-core reentrant:** Unlimited concurrent execution of an interface is possible, including preemption and parallel execution on multi-core systems. This level can be either achieved by mutual exclusion when entering critical regions, or by the absence of such regions, for example if there are no shared resources (including hardware and memory).
- **Single-core reentrant:** Pseudo-concurrent execution (i.e. preemption) of an interface is possible on single core systems. This is the highest level of reentrancy defined by AUTOSAR 4.0.3. Because it does not explicitly cover multi-core systems, “concurrency safe” has been introduced additionally. This level can generally be ensured by the same mechanisms as “concurrency safe”, but they must be ensured to work across core boundaries.
- **Non-reentrant:** Concurrent execution of this interface is not possible.

If a module that is not concurrency safe is invoked in different partitions, there is no warranty that the module will uphold its desired behavior. In this case, correct behavior shall be ensured by the usage of the module, for example if the caller(s) prevent parallel execution by using exclusive areas.

2.3 SchM Interfaces for Parallel BSW execution

This chapter describes the extensions to the SchM required by the concept “Enhanced BSW allocation”.

The Basic Software Scheduler (SchM) is responsible for handling the inter-partition communication between BSW modules. This is conceptually similar to the handling of inter-partition communication between SW-Cs by the RTE. Because the BSW modules are arranged below the RTE in the AUTOSAR architecture however, the communication must be available before the RTE is available. Therefore and for reasons of performance, BSW modules use the SchM for communication.

For the distribution of BSW modules across several partitions, the SchM shall implement the methods `SchM_Call`, `SchM_Result`, `SchM_Send` and `SchM_Receive`, which are used to handle service calls and callbacks as well as writing data to and reading data from a sender-receiver connection. For details on the signatures of these functions, please refer to Section 2.2.3, which describes the SchM extensions from a BSW developer's point of view.

The SchM can use `IocSend` (a direct call to the OS) to send data in inter-partition communication. Other RTE internal mechanism might not be available during startup.

The Inter-OS-Application Communicator (IOC) shall be configured to provide `IocSend_<Id>` functions with a uniquely determined `<Id>` for all client-server and sender-receiver connections that cross partition boundaries.

Analogously, the SchM shall use `IocReceive` to receive data from inter-partition communication, and the IOC shall provide the corresponding `IocReceive_<Id>` functions.

The following frame contains some pseudo code snippets that show how to use the IOC for inter-partition communication.

```
void some_BSW_function(){
  char *str = "some text";
  SchM_Send_Data_Src_DstN(str);
}

Std_ReturnType SchM_Send_Data_Src_DstN(char *str){
  IocSend_1(str, 5);
  ActivateTask(TASK1);
}

Std_ReturnType SchM_Receive_Data_Src_DstN(char *str){
  IocReceive_1(str);
}

TASK(TASK1){
  char data[20];
  SchM_Receive_Data_Master_Sat1(data);

  /* do something with data */
}
```

2.4 Configuration of Basic Software in Partitioned Systems

This is the chapter for integrators.

2.4.1 Task Mapping

The parallelization of BSW modules introduces several new subclasses of BswEvent to the AUTOSAR metamodel. These classes are shown in Figure 4. Each BswEvent (including instances of subclasses of BswEvent) is assigned to a BswSchedulableEntity, which is started upon occurrence of the event.

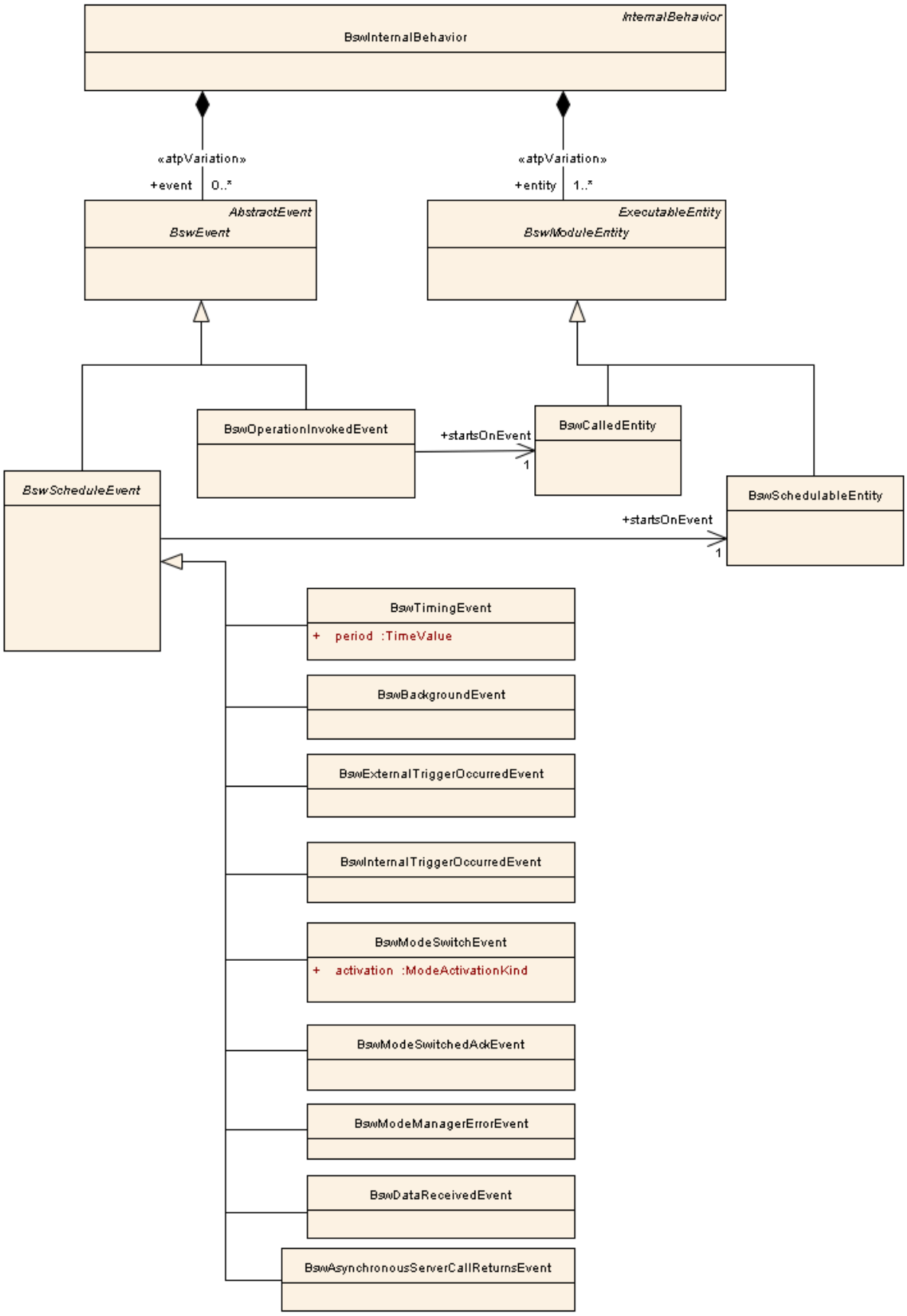

Figure 4: Events triggered by the invocation of BSW functions

A more fine grained description of the partition specific behavior of an entity can be described by the use of BswDistinguishedPartitions, as shown in **Figure 5**. A BswDistinguishedPartition is the abstract representation of a partition, which allows to the mapping of a specific BswEvent, BswModuleCallPoint or BswVariableAccess to a set of abstract partitions. The representation of a partition at this point is an abstract one in the sense that it is part of the BSW module description (according to the module description template), whereas a concrete partition is determined at ECU configuration time.

For example, if a module entity running in partition 1 provides data via a VariableDataPrototype to the same entity running in partitions 2 and 3, the BswModuleEntity aggregates a dataSendPoint with a contextLimitiation to partition 1 and a dataSendPoint with a contextLimitation to partitions 2 and 3.

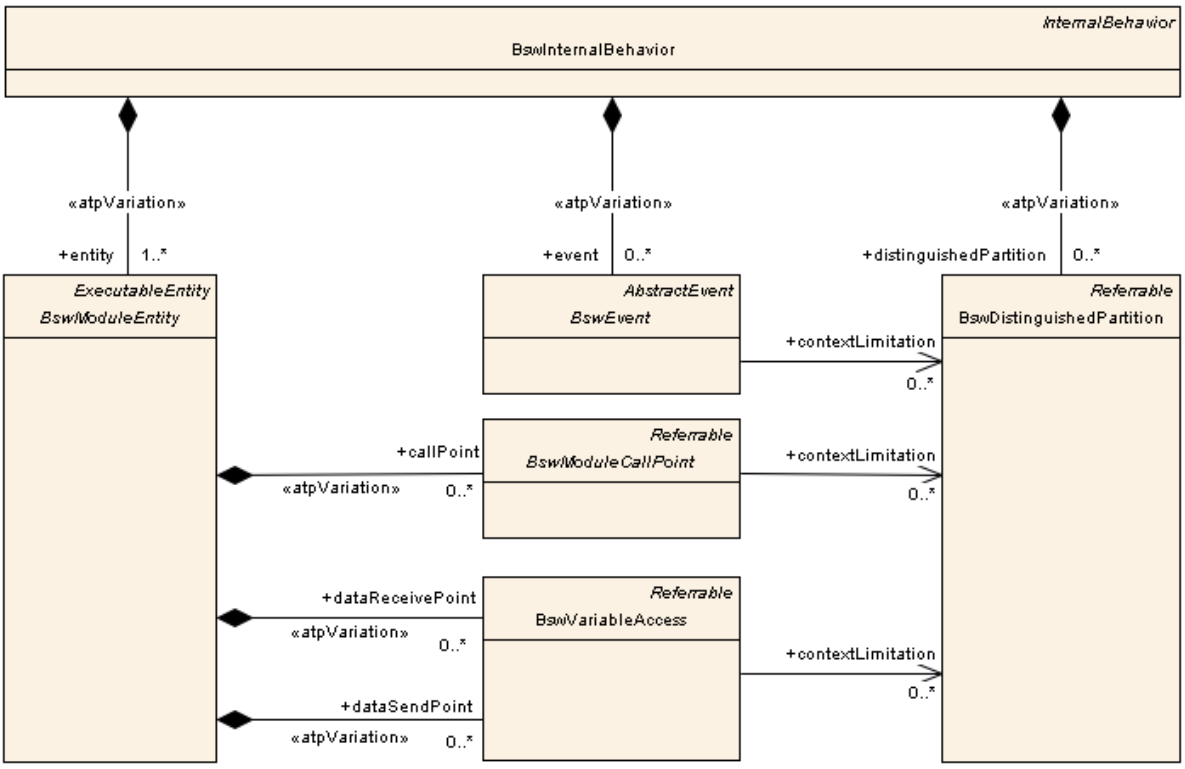

Figure 5: Modeling partition specific properties of entities using BswDistinguishedPartitions

The actual partition for the handling of an event is determined by its task mapping.

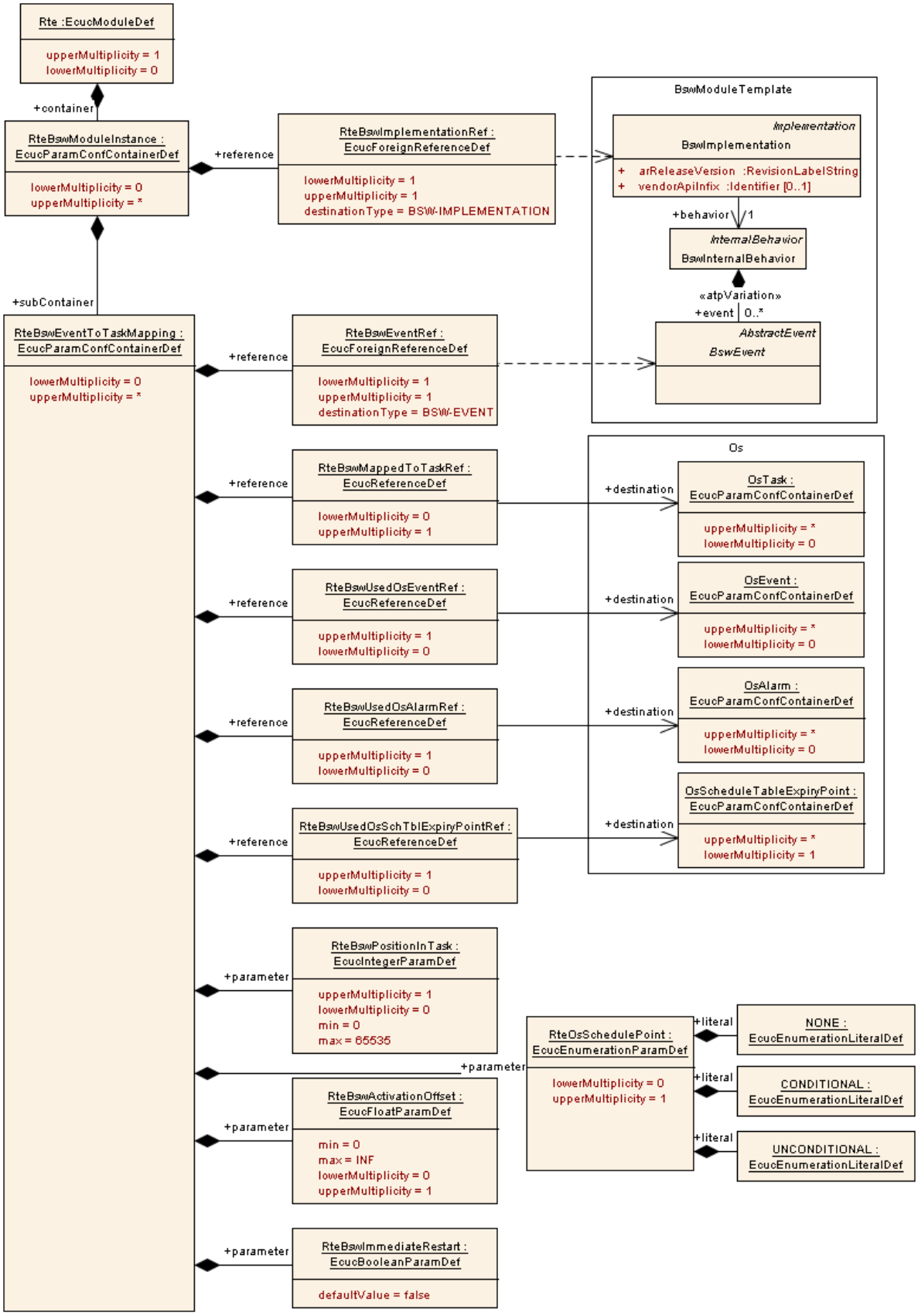

Figure 6 shows the corresponding excerpt from the AUTOSAR metamodel.

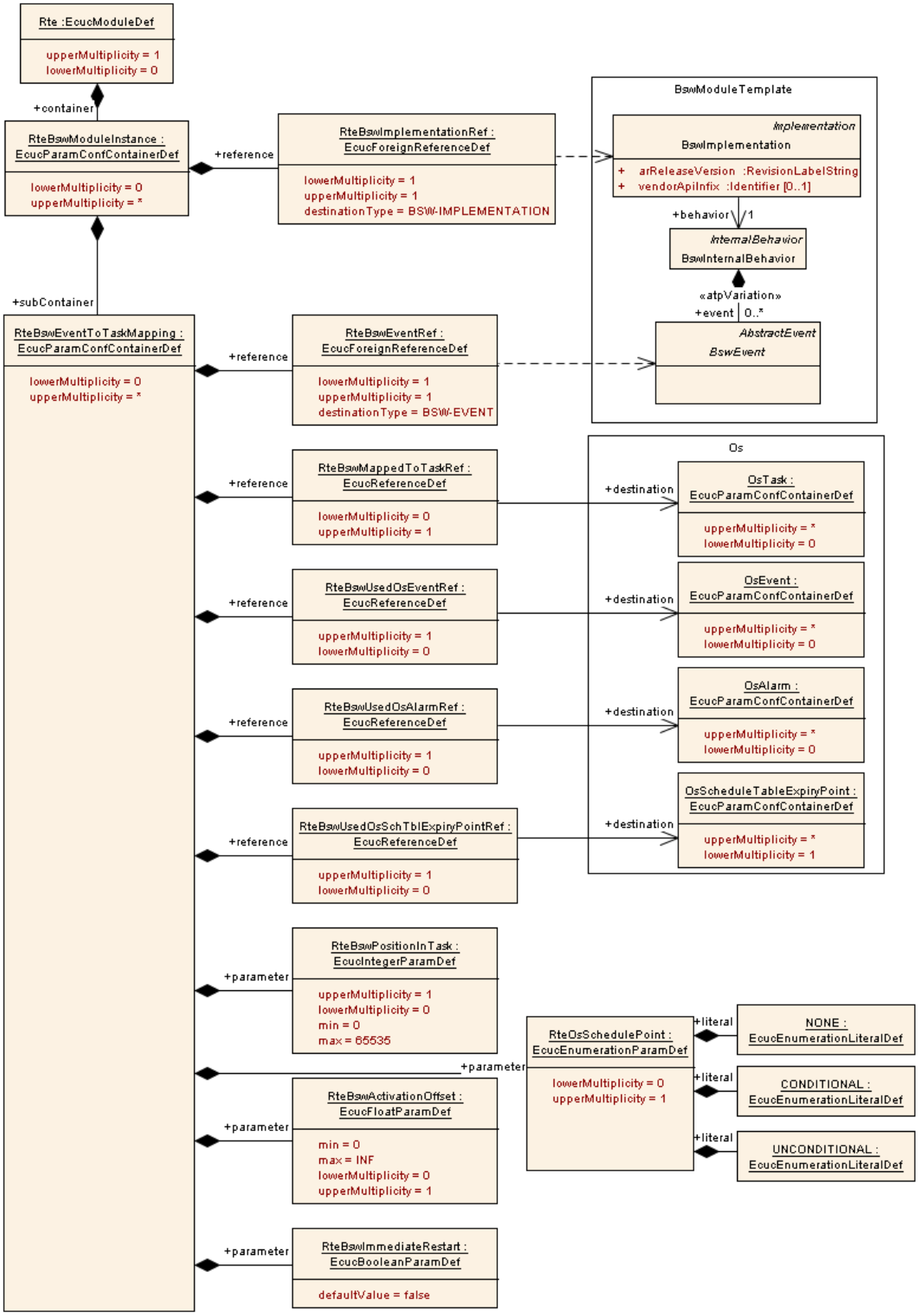

Figure 6: Mapping OperationInvokedEvents to tasks

An RteBswEventToTaskMapping refers to a BswEvent (indirectly via its RteBswEventRef) and to an OsTask (also indirectly via its RteBswMappedToTaskRef). The task is in turn mapped to a partition, and the partition is mapped to a µC core, which is the core responsible for the processing of the event. Mapping an event to a task is optional; if an event is not mapped to a task, it is handled in its originating partition. If no special mechanisms apply that prevent concurrent execution, a prerequisite for a non-mandatory mapping of an event to a task is:

- if the BSW entity is shared between multiple BSW partitions the entity needs to be *concurrency safe*
- in case it is exclusively available only on one BSW partition it needs to be at least *reentrant*.

Please note that it is currently not allowed to map RunnableEntities of a SW component to multiple partitions [SWS_Rte_07347]. For BSW it is possible to map the same module entities to different tasks and partitions by using different BSWEvents referring to the same entity

2.4.2 General Configuration of Master and Satellites

Modules that shall be available in multiple partitions can be implemented as masters and satellites. In this case, the master and all satellites of the same module share the same code (which may implement core-dependent behavior however) and the same configuration. Hence, a master and its satellites are treated as one module entity w.r.t. their configuration.

The communication between master and satellites is not to be standardized. It is considered to be module-internal and it is not visible to other modules. However, since it is recommended to use SchM mechanisms for internal communication, the non-standardized client-server entries and data accesses in the BSWMD to connect master and satellite need to be configured.

2.4.3 Configuring the BswM (per Partition)

On systems with distributed BSW there is one BSW Mode Manager (BswM) per partition (but one OS and EcuM per core, which is the same as long as we have one BSW partition per core). Each of these BswMs can be configured independently. A BswM mainly interacts with the state managers (ECU state manager and bus state managers, for instance) on the same partition.

The BswM is also responsible for the initialization and shutdown of BSW modules running in the same partition. Therefore, its configuration depends on the mapping of BSW modules to partitions.

The configuration of the BswMs is split across the container BswMGeneral, which contains shared configuration parameters of all BswM entities and BswMConfig containers, where one BswMConfig is defined for each BswM entity. Consequenty, the mapping of a BswM to its partition is defined in the corresponding BswMConfig container, which has a BswMPartitionRef pointing to the respective partition. This mapping of BswM configurations to partitions ensures that for every partition the correct configuration of the BswM can be determined.

Additional extensions to the BswM configurations for the allocation of BSW modules to multiple partitions are

- A reference BswMRequestRemoteMode in the container BswMAvailableActions. This action indicates a call to a BswM in a different partition, which is used to propagate mode requests.
- References BswMBswMModeRequest and BswMBswMModeSwitchNotification in the container BswMModeRequestSource. The BswMBswMModeRequest indicates that the source of a mode request is a BswM running in a different partition (ECUC_BswM_00980, cf. [5]). BswMBswMModeSwitchNotification indicates that another BswM has switched a mode.
- All functions listed in an action list that is processed by a BswM entity must be available in the partition this BswM is running in.

2.4.4 Configuring the EcuM (per Core)

On systems with distributed BSW there is one EcuM per core (even if there are multiple BSW partitions on that core). In other words, on every core there shall be one and only one partition that runs the EcuM. The partition running the EcuM is determined by the EcuMFlexEcucPartitionRef, which is specified in the container EcuMFlexUserConfig of the EcuM configuration.

Distributing the BSW is only possible when using the EcuM Flex; the EcuM Fixed does not support this.

On architectures with a sequential start of cores, there is one designated master core in which the boot loader starts the master EcuM via EcuM_init. The EcuM in the master core starts some drivers, determines the Post Build configuration and starts all remaining cores with all their satellite EcuMs.

On architectures where all cores are started at the same time, core dependent branching within the EcuM_init function can be used to achieve core-specific behavior. This can in turn be used to identify the EcuM master (running on the master core), which is responsible for the EcuM initialization on the slaves.

3 BSW Distribution in Safety Systems

3.1 General overview on safety

In today's cars several ECUs may control safety relevant actuators depending on the functionality of the vehicle. Examples are electronic steering lock systems, adaptive cruise control systems or braking systems. If such a system shows a misbehavior a dangerous situation can occur where the driver is no longer able to drive the car in a save manner. To avoid such failures the specific ECUs must be developed in a way that the system can detect and react in a controlled way to such faults. The ISO 26262 is the norm which describes how the development of such ECUs shall be performed to realize a save system. This norm defines four "Automotive Integrity Safety Levels" (ASIL) which classify the risk of the system. Based on the risks specific (safety) requirements of the system are derived. These requirements may be related to hardware (e.g. support for multiple channels to allow detection of hardware problems) or software (e.g. control flow checking) or both. In AUTOSAR we focus on software, so the hardware part will no longer be considered here. Be aware that an ASIL is always defined for a system, which means hard- and software, and with respect to software application software and basic software.

3.2 Safety solutions in AUTOSAR

AUTOSAR up to R4.1 supports safety systems by offering different base mechanisms which are typically required in such ECUs. The following list contains the main safety mechanisms:

- Partitioning of SWCs to support the isolation in space. This means that it is possible to separate SWCs of different ASIL from each other and to make sure that the SWCs are not able to write to other SWCs data. The realization requires hardware support (a memory protection or memory management unit) and is realized in the Os module and used by the Rte.

- Timing and control flow supervision to monitor executing entities and to detect faults caused by blocking or wrong execution. In AUTOSAR the Os and the WdgM take care of this issue.

- A safe communication via end-to-end protection is possible between ECUs (and even inside an ECU). This guarantees e.g. that the data which is send is not modified between the sender and the receiver(s). The responsible module is the E2Elibrary.

Some other modules support additional mechanisms which are also useful in safety systems (e.g. CoreTest or RamTest).

The following picture shows how an AUTOSAR R4.1 can be used to support an ASIL ECU.

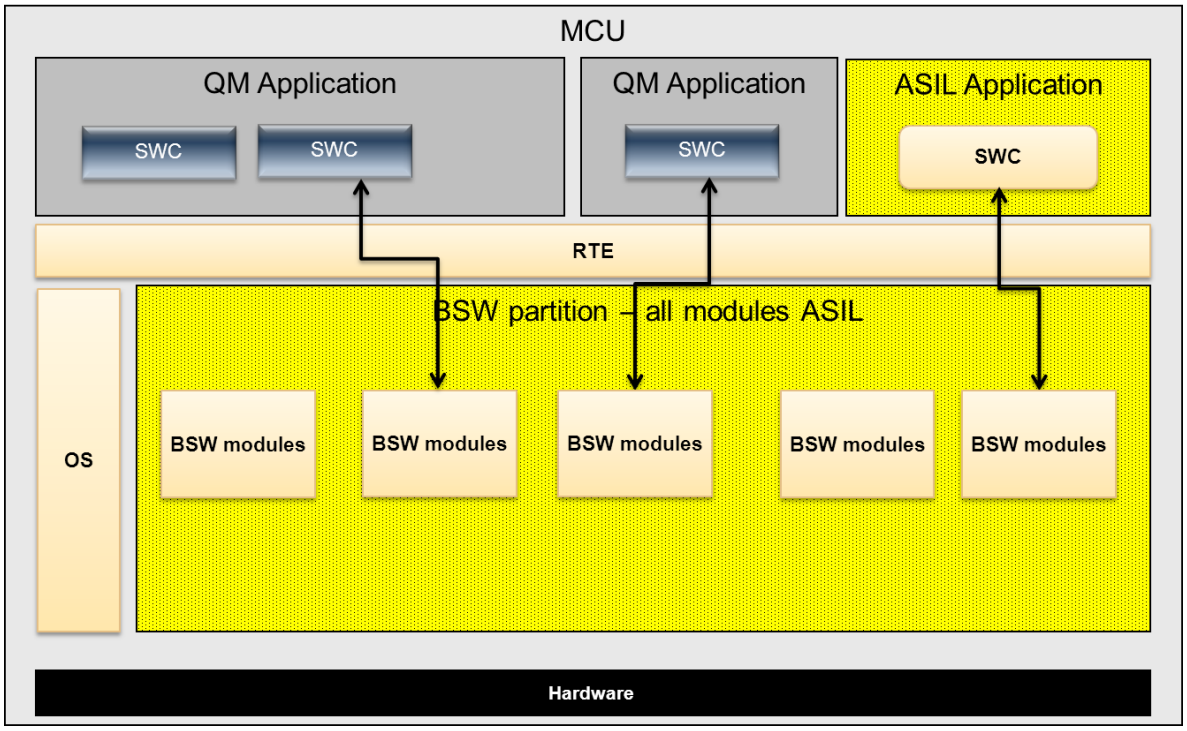

Figure 7: All BSW developed according ASIL

The approach works but has one big disadvantage: **all** BSW modules must be developed according the highest ASIL of the system. This causes a lot of additional work even if only some of the BSW modules are really required for a specific safety requirement.

Starting with R4.2 AUTOSAR offers an additional way how a safe system can be developed without the requirement to implement the whole BSW with the according ASIL. The key aspects of the new approach are:

- The BSW modules are not all mapped to one partition, but can be placed in separate partitions depending on the ASIL need. This means that a system can have one QM partition and a partition for each ASIL level (or even more ASIL partitions)

- The impact of the approach to single BSW modules is minimal. This means the scope of the modules is the same on ASIL and QM. There is no change of interfaces between modules.

- Only the modules which provide the safety relevant features (e.g. the memory protection offered by the Os) need to be developed according to the system's ASIL. Sometimes it is even possible to limit the required ASIL functionality to a subset of a BSW module.

The ASIL modules inside the ASIL partition(s) need to be specifically developed. They not only need to meet the requirements of the ASIL level, but they also need to detect if they are called from inside the partition or outside the partition.

With this approach it is possible:

- To reuse existing BSW modules which were developed on QM level (no ASIL) without module modification.

The proposed approach has to be assessed case by case in order to estimate the applicability of this approach for the particular safety case and the benefits of combining QM/ASIL modules compared to a pure ASIL approach.

BSW modules can be placed in different partitions. AUTOSAR supports one QM partition and several ASIL partitions. The following figure shows an example mapping. Here the ASIL SWC has save access to some hardware via an own partition in the BSW which contains an IoHwAbs and the needed drivers below.

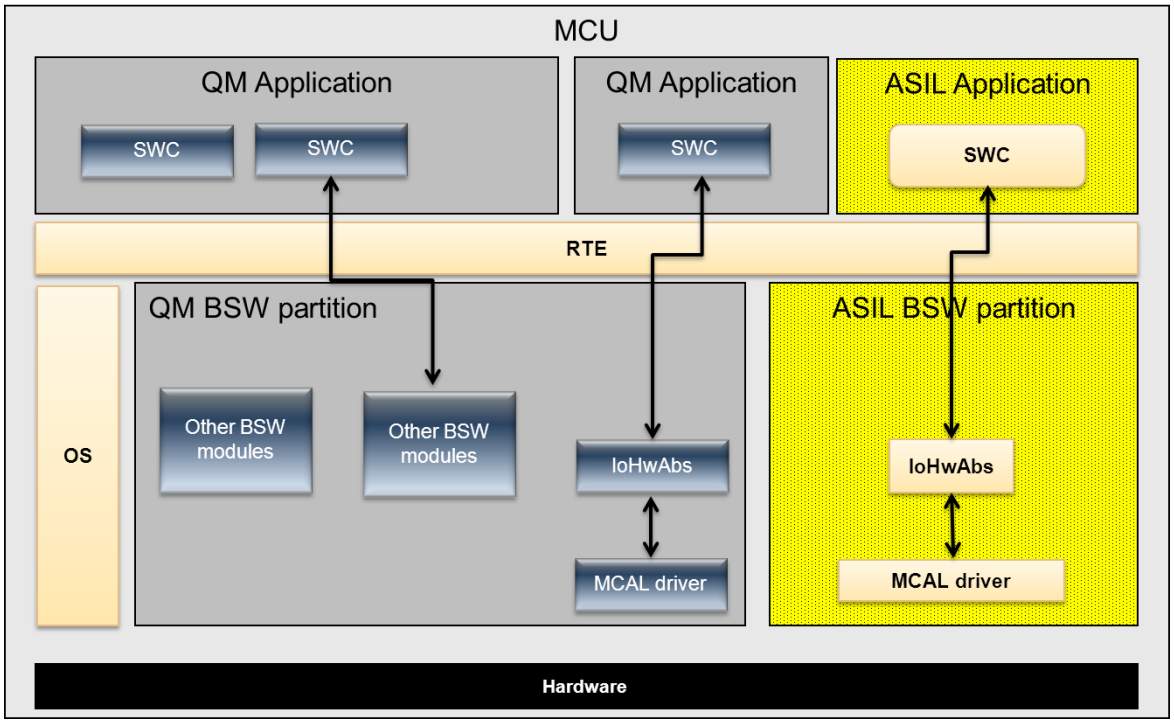

Figure 8: BSW modules mapped in different partitions

It is strongly recommended that QM BSW partitions run in user mode if possible in case we have BSW ASIL partitions in the system to avoid changes to hardware registers (e.g. MPU settings). If this is not possible (e.g. hardware supports supervisor mode only) you need additional means to assure freedom from interference.

3.2.1 Some modules are always ASIL

Since the protection mechanism is provided by some specific BSW modules (e.g. the Operating System) these modules have to be developed according to the highest ASIL in the system. If they are not developed at this level it cannot be assured that they are able to fulfill their supervision task. The decision which modules have to be

developed to ASIL is always project specific and is determined from the safety requirements of the system.

3.2.2 Overall configuration

The separation of BSW modules in different BSW partitions for safety needs to be configured in the ECU configuration. The mapping is done in the EcuC and Os configurations.
For each such BSW partition an OsApplication is required. The following settings apply to the Os configuration of each BSW OsApplication:

Name	Value for BSW partitions
`OsTrusted`	`TRUE`
`OsTrustedApplicationWithProtection`	`TRUE` or `FALSE`
`OsTrustedApplicationDelayTimingViolationCall`	`TRUE`

Other attributes of the OsApplication can be filled as needed. Note that hook functions of BSW partitions have no meaning in AUTOSAR and shall be avoided.

Additionally note that the OSApplication TRUSTED attribute (`OsTrusted`) of the OS-Application is not related to ASIL/non-ASIL.

Afterwards the BSW modules, which are used, have to be configured and mapped to the different partitions. The mapping is done in EcuC:

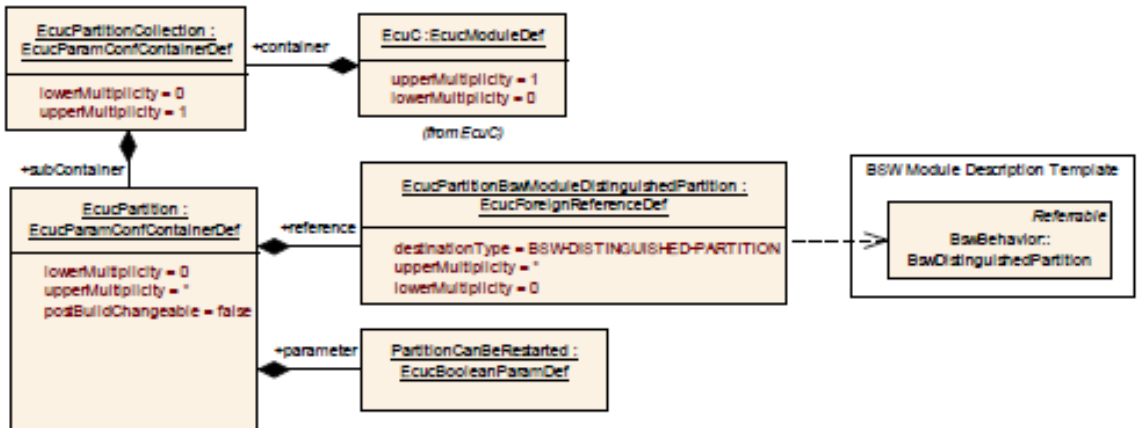

Figure 9: EcuC configuration – mapping of BSW to partitions

The `EcucPartitionCollection` (multiplicity 0..1) contains all partitions of the system. For each of them a sub container `EcucPartition` (0..*) exists which contains references (`EcucPartitionBswModuleDistinguishedPartition` (0..*)) to the BSW modules (via BSWDT) which are placed into this partition.

The following settings apply to the `EcucPartition` configuration of each BSW partition:

Name	Value for BSW partitions

EcucPartitionBswQmModuleExecution	TRUE for QM modules FALSE for ASIL modules
PartitionCanBeRestarted	FALSE
EcucPartitionBswModuleExecution	TRUE
OsAppEcucPartitionRef	Link to the OsApplication of this partition

At the end we have one QM partition and one (or more) ASIL partitions configured

3.2.3 Crossing partition boundaries

When BSW modules are placed into different partitions, the crossing of boundaries is the biggest issue which needs to be solved. The following figure shows the scenario in a quite general view:

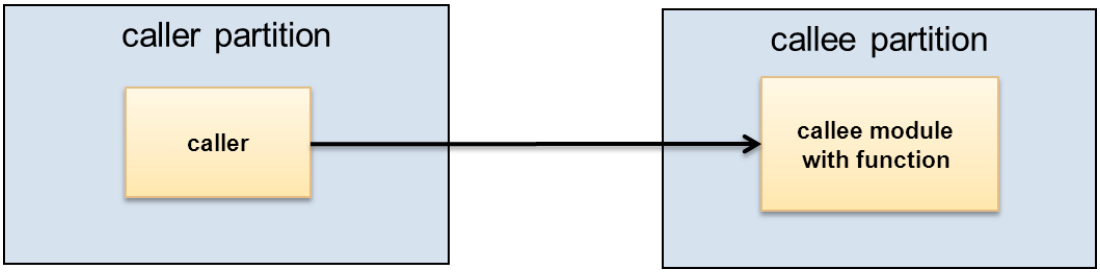

Figure 10: Cross partition call

This is due to the fact that the called service assumes that it has full access to module local data, which is not true if the call is performed from another partition because the memory protection settings are still those of the caller. In general there are 3 possibilities how the problem can be solved:

1. Instead of a direct call the caller can do an ActivateTask() to a Task from the callee partition. In this case the activated Task will perform the real call to the function. Instead of the ActivateTask() a SetEvent() can be used as an alternative. Note that both mechanisms work in an asynchronous way which means that the original caller may need to wait or have to poll for the result

2. The caller can use CallTrustedFunction() to enter the callee partition, or the callee after being called use CallTrustedFunction() to hand over to its partition. After entering the function can be called directly. CallTrustedFuntion() makes sure that the caller gets the appropriate rights to make the call, e.g. changing the memory protection to the setting of the called function.

3. The call of the function may be directly possible if the called function does not write to own data or calls other functions which write to such data. E.g. if the function just reads out a value and return it. Basically, such a function behaves like a library.

Dependent on the mapping of the BSW modules to different partitions the right option has to be chosen. For all function calls between BSW modules located in different partitions which are synchronous, we will focus on the calling possibilities (2) and (3). Because as already stated QM modules are not changed, we have to encapsulate calls which are made from QM partitions to ASIL and vice versa. The ASIL module is always responsible to handle the boundary crossing since the QM module is not touched and does not know this border. This means that if the ASIL module is the caller, the boundary handling needs to take place on the caller side, and if the ASIL module is the callee, the boundary handling needs to take place on the callees side

The following descriptions focus on ASIL and QM BSW modules. Besides BSW modules also CDD might be included in the system. For CDDs the same rules and restrictions apply (if not otherwise explicitly stated)

3.2.3.1 QM modules calls ASIL

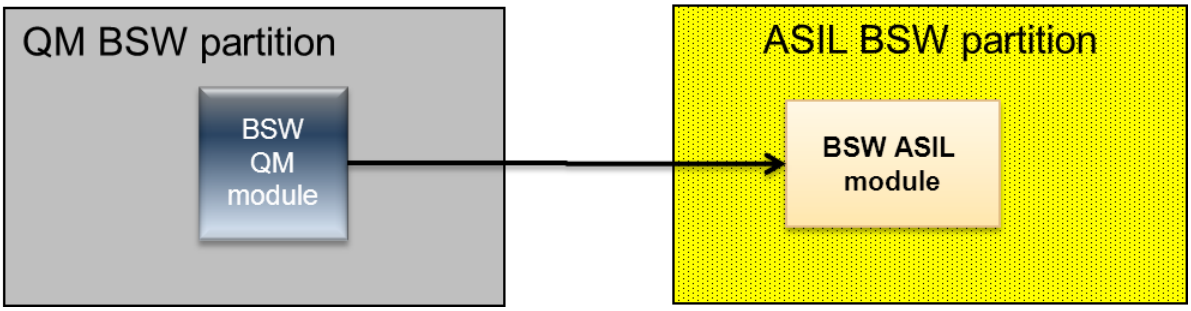

Figure 11: QM calls ASIL

As already stated the QM module which performs the call is unchanged. Even more: The QM not even knows that the called function (module) belongs to a different partition. This means we have to encapsulate the called function into a stub which performs the boundary crossing.

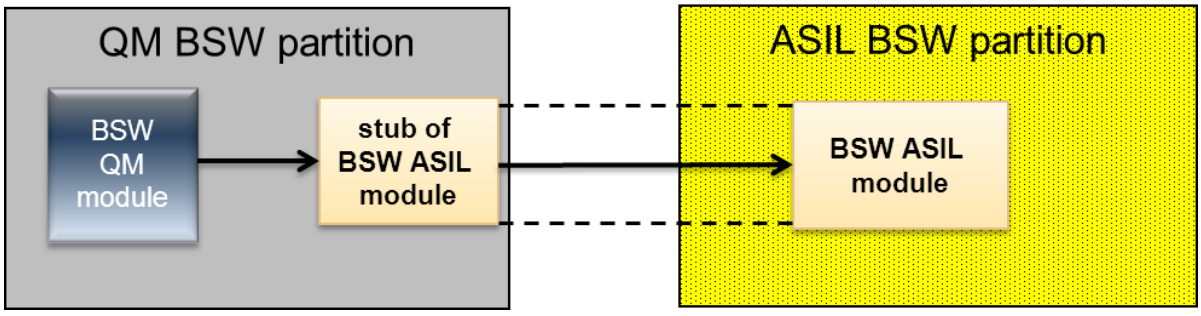

Figure 12: Details of QM calls ASIL

This stub function can be static or generated and belongs to the called module. It can be seen as a new function entry of the called function of the ASIL module. The following message sequence chart shows the calling sequence. As you can see the stub itself also has two parts, one on the caller side and one on the callee partition.

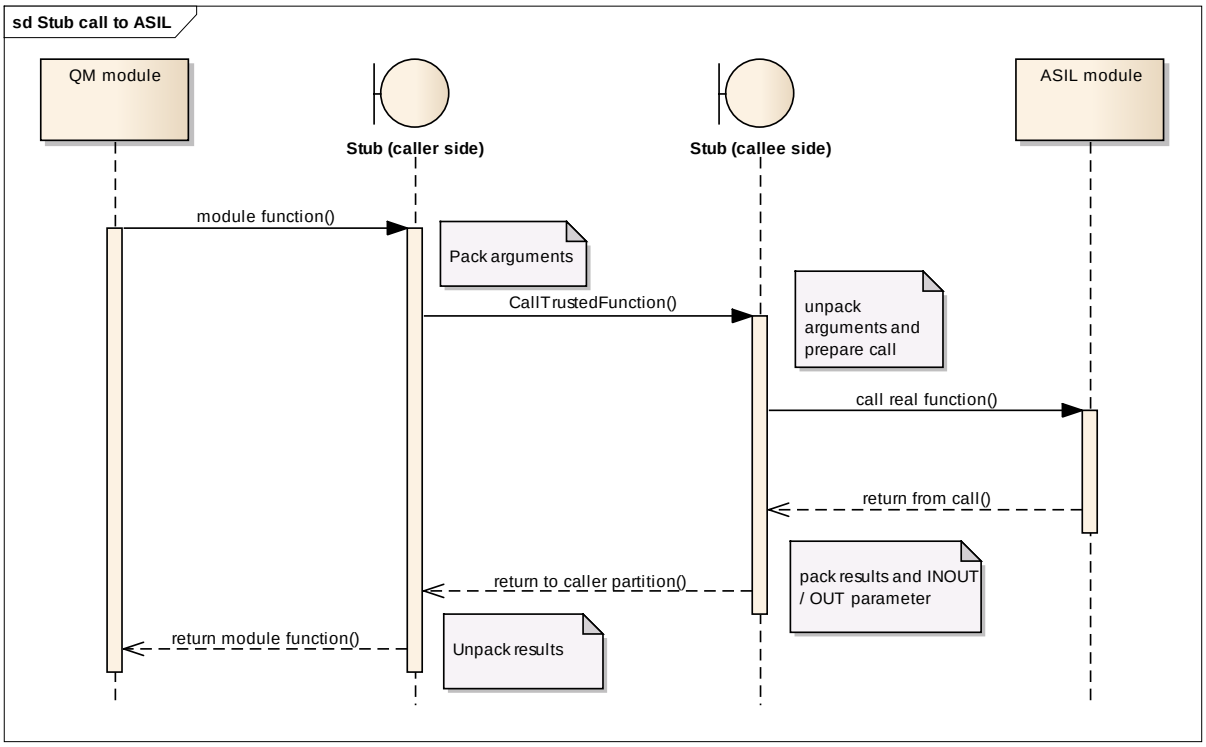

Figure 13: Call sequence when a stub is used

The stub itself can be static (hand written) or generated based on the available configuration information. The next two sub chapters are detailing the different approaches.

3.2.3.1.1 Static stub

A static stub has to cover all situations. In our case the important issue is to find the caller partition in order to make type of call. The next code fragment shows an example of a static stub:

```
StdReturnType module_function()
{
   runId = GetCurrentApplicationId();
   if (runId == module_applicationId)
   {  /* direct call possible */
      return Modulemodule_function_real()
   } else {
      CallTrustedFunction(MODULE_REALFUNCTION_ID,NULL)
     …
   }
}
```

Note that you have to init your own module application Id (or use directly the generated application name)

3.2.3.1.2 Generated stub

If an optimized version of the stub shall be generated the generator needs all information (e.g. who calls the function) in order to create the best code. If information is missing or incomplete the generated code might either not be able to generate the code at all or the code may fail during runtime.
AUTOSAR has an abstraction for calls between different partitions. This method is used in multicore systems to allow modules a communication between different partitions on different cores.

The mechanism used by the generated code is offered by the SchM: SchM_Call(). The SchM_Call() will then be mapped within the SchM to one of the methods listed in 3.2.2.

For finding the best method for crossing the boundary the central question is:

Who will call the function (and use the stub)?

This information must be provided by the user via the SchM configuration. The configuration consists of caller, callee and references to their modules (and also implicit to the partitions). The following diagram from the RTE shows the configuration of SchM_Call():

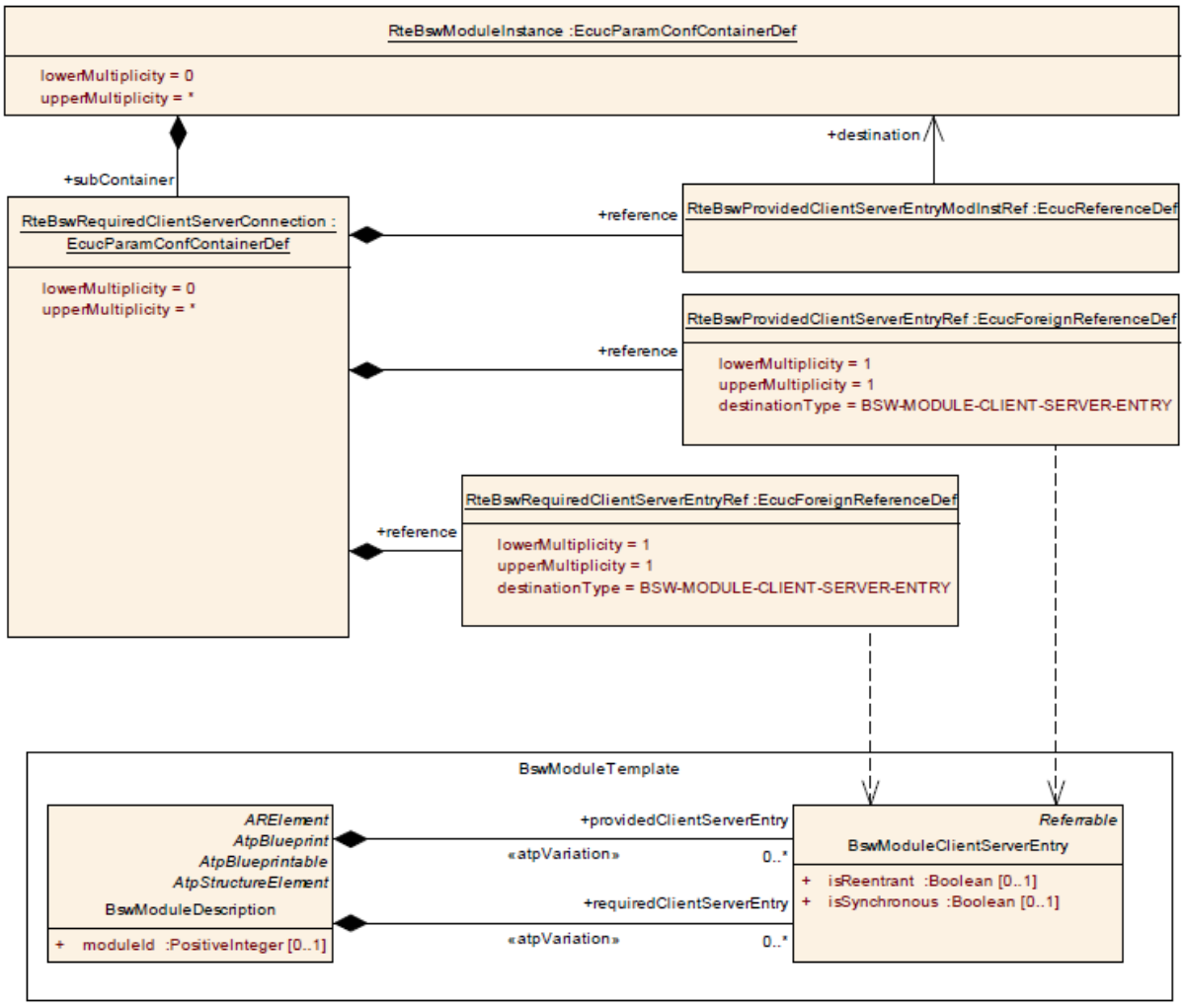

Figure 14: Configuration of SchM_Call()

Based on this information and the information where a BSW module is placed, the SchM can generate optimized version of the SchM_Call().
E.g. if there is only one stub user and this user is placed in the same partition as the called BSW module a direct call is possible. Example of a stub using SchM_Call():

```
Std_ReturnType  module_function()
{
   Std_ReturnType r;
   (void) SchM_Call_target_module_function(&r);
   Return r;
}
```

The approach to generate a stub has some limits which need to be considered during system development:

- Calls from integrator code: A configuration via SchM_Call() is not possible for integrator code since this code does not belong to any BSW module and does not have any configuration (EcuConfiguration) and module (BSWDT) information which could be used. In such cases a hand written static stub has to be used.

- A SchM_Call() configures exactly one caller-callee relationship. If a function is called by different callers, the generated part of the stub cannot distinguish which SchM_Call() is required for which caller. In such cases a static stub is required.

Note: If also the QM caller would use a SchM_Call() instead of the real function name the stub could be avoided completely. But this would contradict the target to reuse existing QM code unmodified.

For parameter handling see 3.2.3.5.

3.2.3.2 ASIL calls QM partition

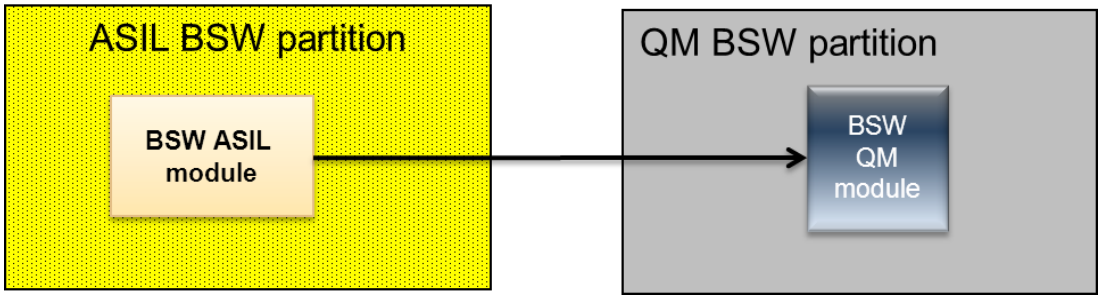

Figure 15: ASIL calls QM

This chapter covers now the direction of an ASIL caller and a QM callee. Here the ASIL module already knows that a boundary crossing is required. (Otherwise the called QM function would be an ASIL function). Since the QM function shall not detect any difference when called from an ASIL function or from a QM function in the same partition, it must be called as would the call be locally performed.

As a consequence of this we need again a code fragment which performs the real call. This code fragment in this case is named wrapper.

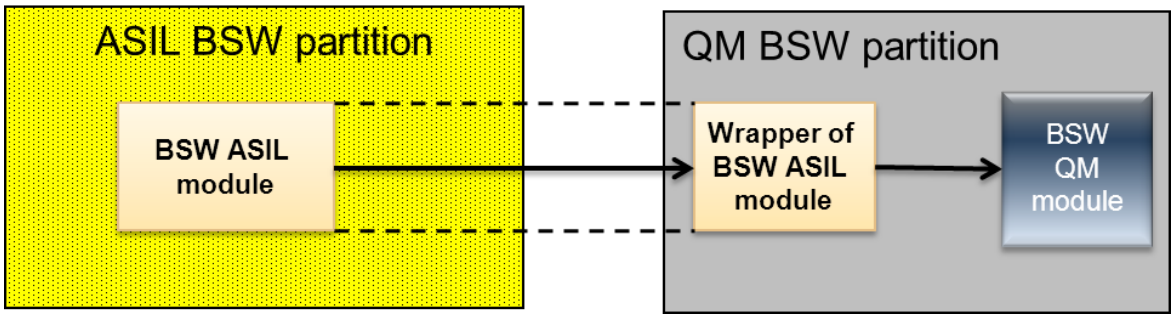

Figure 16: Wrapper for ASIL calls to QM

This wrapper function can be statically or dynamically generated and belongs to the caller module but is partly executed in the partition of the callee. The following message sequence chart shows the calling sequence, when CallTrustedFunctio() is used:

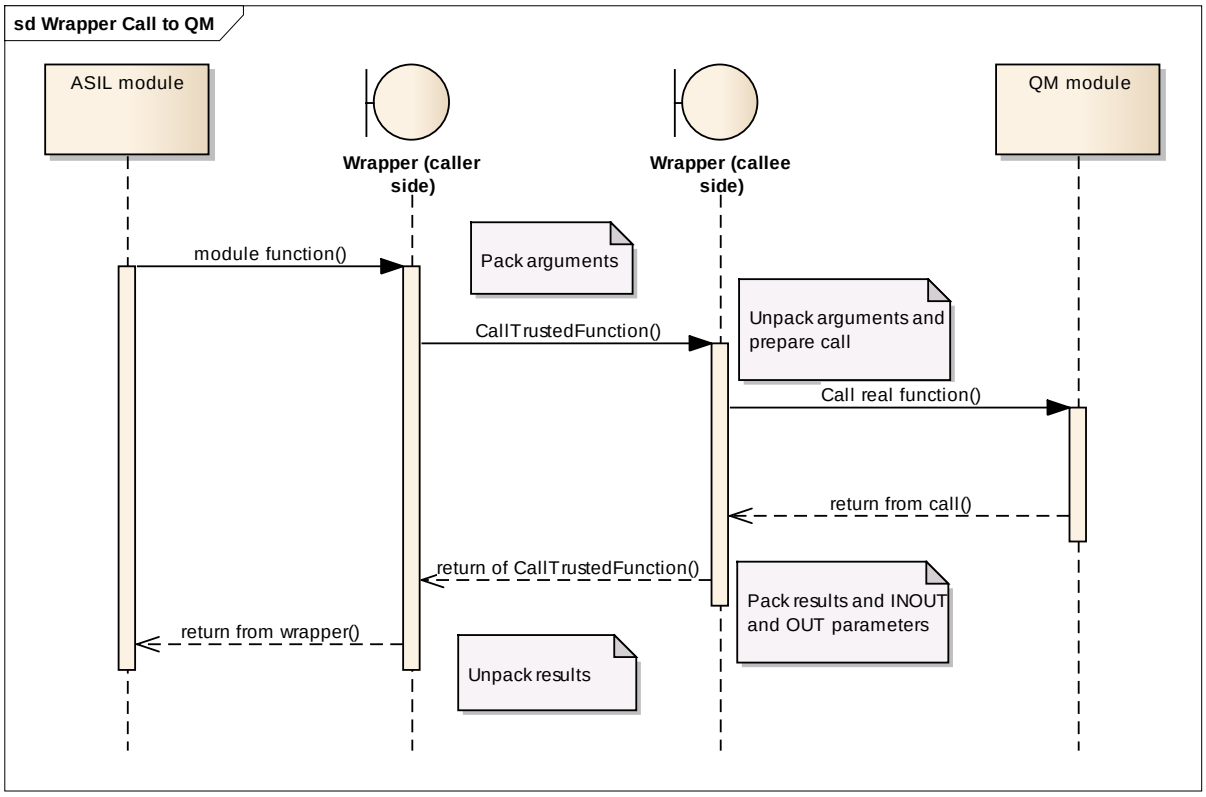

Figure 17: Call sequence when a wrapper is used

We can again differentiate in a static wrapper and wrappers which are generated out of the configuration.

Note that independent of the technical solution it needs to be checked whether such calls are allowed within the project specific safety goals.

3.2.3.2.1 Static wrapper

The following code fragment shows a possible wrapper in case only one “user” calls the function (in other cases the buffer handling needs to be extended).
In the example the CallTrustedFunction() mechanism is used:

```
uint8 wrapper_function()
{
    /* … */
    CallTrustedFunction(MODULE_REALFUNCTION_ID,NULL)
    return function_return_value;
}
```

This is the second part of the wrapper which is located in the callee partition:

```
uint8 function_return_value;

void TRUSTED_call_function (TrustedFunctionIndexType a,
                        parameter_struct *local_struct)
{
    function_return_value = function();
    return;
}
```

3.2.3.2.2 Generated wrapper

If the wrapper shall be generated the generator needs specific information in order to create the best code. If information is missing or incomplete the generated wrapper code might fail.
Like the stub handling In 3.2.3.1 we can use the SchM_Call() service to hide the partition transitioning. In contrast to the stubs we need not to focus on possible users of the wrapper – the users are just the ASIL module functions – but on the called function. This means we have to find out the callees partition in order to make the right call. Since we only support one QM partition, we can just look this up (parameter `EcucPartitionBswQmModuleExecution` ist `TRUE`) and know where the call must be performed.

There is also one limitation of this approach:

- Calls to integrator code: A configuration via SchM_Call() is not possible since the integrator code does not belong to any BSW module and does not have any configuration (EcuConfiguration) and module (BSWDT) information which could be used. In such cases a separate static wrapper has to be used to encapsulate calls from integrator code and the integrator code need small changes, e.g. changing the name of the called function to avoid name clashes.

For parameter handling see 3.2.3.5.

3.2.3.3 ASIL calls ASIL

The case of an ASIL to ASIL call can be seen as a combination of 3.2.3.2 and 3.2.3.1. Also here a generic glue code might be needed if the modules are not placed in the same ASIL partition. In this case either the caller or the callee have to provide this glue code. In an ASIL system the glue code is normally provided by those modules which have the higher ASIL. The glue code can be created statically or can be generated.

For the generation of the glue code the following limitations exist:

- Calls to integrator code: A configuration via SchM_Call() is not possible since the integrator code does not belong to any BSW module and does not have any configuration (EcuConfiguration) and module (BSWDT) information which could be used. In such cases
    - Either a static glue code has to be used to encapsulate calls from/to integrator code and the integrator code might need small changes, e.g. changing the name of the called function to avoid name clashes.
    - or offer vendor specific configuration parameter which holds per callout a reference to the OsApplication where the integration code is placed.

- If we know only the address of the callee (this can happen if the interface is generic and function pointers are used for the call, e.g. in the PDU Router) we need a dedicated vendor specific configuration parameter for the ASIL module which provides the information in which partition the callee is located.

3.2.3.4 QM calls QM

This caller-callee combination is not supported by AUTOSAR. Reason is that this is not possible without changing an existing QM module. Therefore, only one BSW QM partition is supported. Hence all these calls are partition local.

3.2.3.5 Parameter passing

In the previous sections we showed how a call to a function in another partition can be made. Besides the real call mechanism there is another important topic and this is the passing of parameters to the callee and passing results back to the caller. The question behind this is: How does the callee access these parameters and how can results be propagated back to the caller.

AUTOSAR differentiates between IN, OUT and INOUT parameters which are passed. IN parameters are not critical, because they are normally passed by value and even for cases where a by reference passing is done the callee is not allowed to write to them. This means that they do not pass any information back to the caller. OUT and INOUT parameters are used to return results from the callee back to the caller. The question now is: how can these values passed back to the caller if callee and caller are not in the same partition.

In general the following methods are possible:

1. If caller and callee are in different partitions the callee works on a copy (for INOUT data) or empty space (OUT data) and when returning back to the caller the values are copied back. For the inter partition communication of data AUTOSAR offers the IOC mechanism of the Os. However, often usage of IOC can be avoided by copying such that only read access is needed.

2. A hardware specific solution: In such cases a copy / extra buffer is avoided by using dedicated hardware features of the used microcontroller which guarantee freedom of interference. E.g. If the hardware allows for private shared memory areas between caller and callee.

In the following we will show how (1) works. Option (2) depends on the used hardware and is not standardized in AUTOSAR. The following code fragment shows an example how the parameter passing works (case: ASIL calls QM)

```
/* caller side code */
Std_ReturnType _Dem_GetOperationCycleState(
    uint8 id,
    Dem_OperationCycleStateType* state)
{
     ….
     /* setup params struct with arguments */

     ret = CallTrustedFunction(GETCYCLESTATE,&params)
     if (ret == E_OK)
     {
         IocReceive_RETURNVALUEGETCYCLESTATE(&ret);
         IocReceive_VALUEGETCYCLESTATE(state);
     }
     return ret;
}
```

```
/* callee side code */
void TRUSTED_GETCYCLESTATE(TrustedFunctionIndexType a,
                           parameter_struct *local_struct)
{
    Std_ReturnType localreturn;
    uint 8         localid;
    Dem_OperationCycleStateType localstate

    /* setup parameters from local_struct */
    …

    localreturn = Dem_GetOperationCycleState(localid,
                                             &localstate);
    IocSend_RETURNVALUEGETCYCLESTATE(localreturn);
    IocSend_VALUEGETCYCLESTATE(localstate);

    return;
```

Note that the above example is quite typical for AUTOSAR inter-partition calls. It assumes that the lifetime of the buffer is equal to the duration of the called function. If this is different, e.g. one function which just provides a buffer and another function at a later time indicate that the buffer is now ready (example: NvM read mechanism) an adoption is needed.

3.2.4 Access to peripherals / hardware

In AUTOSAR the access to peripherals or hardware is limited to BSW modules. Typically only some of them require a real access, e.g.:

- The Os switches between different contexts and need to read/write the context registers. Also interrupt locking requires normally access to hardware registers or execution of privileged instructions.
- During startup the Mcu driver needs to enable the microcontroller clocks and may perform further initialization of registers
- IO drivers need to access their part of the hardware.
- …

If parts of the BSW are now running in a partition where the memory protection is enabled the full access to hardware is normally no longer possible. In such cases a hardware access can be realized by:

1. “CDD approach”: Create a piece of code which access the required hardware and map this code to a trusted OsAppication with memory protection disabled. This allows the code to have full access. From within your BSW module all hardware access must then call this small piece of code. In this case this code has full access to hardware.
2. “Hardware approach”: If possible map the hardware registers into the address space of the partition which requires the access. This normally opens the access to these registers for the BSW modules which are located in the partition. The availability of this method depends strongly on the used microcontroller and the capabilities of the memory protection unit.

Example for the “CDD approach”: A CDD offers methods to read (peek) and write (poke) hardware registers. Note that in such cases it should be mentioned that additionally an access management is necessary (“Who is allowed to call these functions?”) because otherwise you could not guarantee freedom from interference. The CDD is mapped to an partition with full memory access.

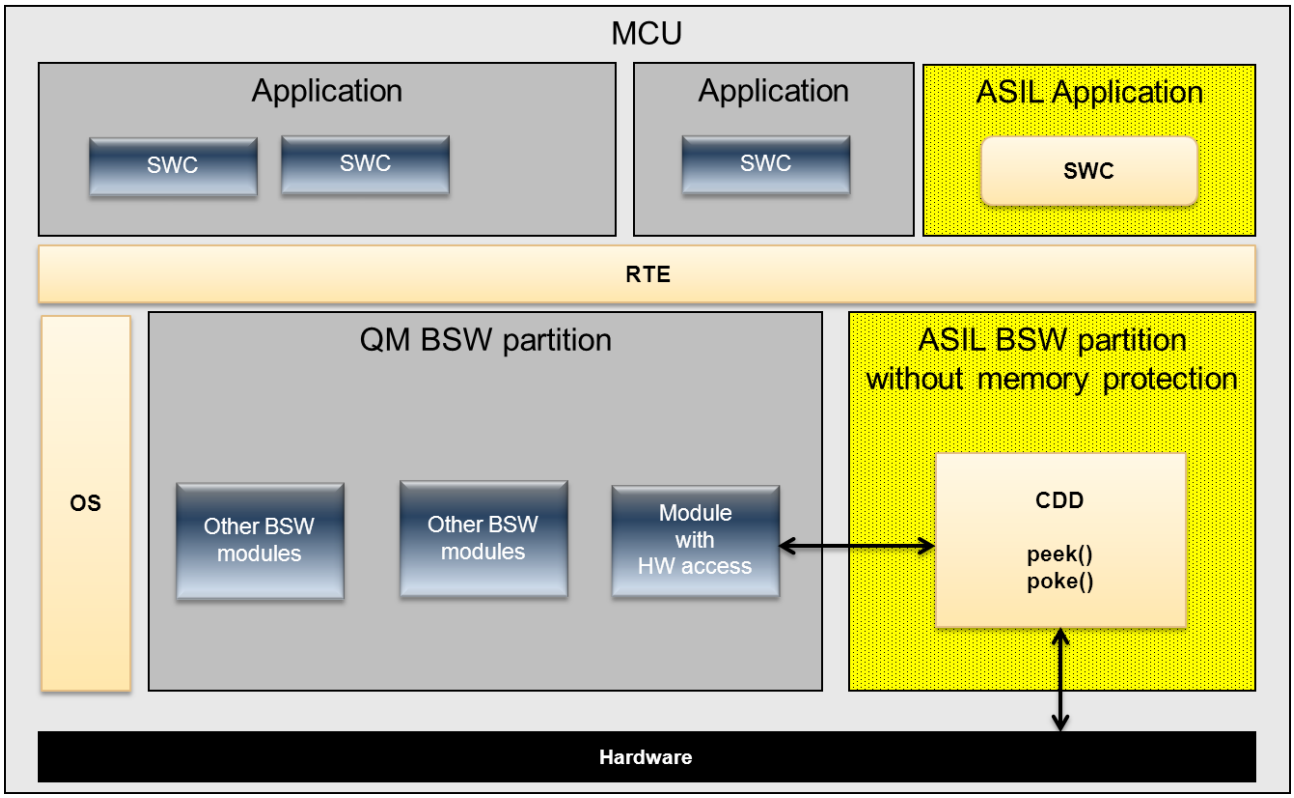

Figure 18: CDD approach

Note that some modules typically have implicit access, because their code is executed before the memory protection scheme is started in the Os. Details can be found in the next chapter.

3.2.5 Startup, Shutdown and Sleep/Wakeup

3.2.5.1 Startup

In AUTOSAR the startup is handled by the EcuM module. It takes care about the right order during system start. In an ASIL system the user has to take care that during startup no relevant data is overwritten or the issue is at least detected. Such faults can happen because the memory protection is not yet running because the Os is not yet started. The following figure from the EcuM shows the default sequence during startup.

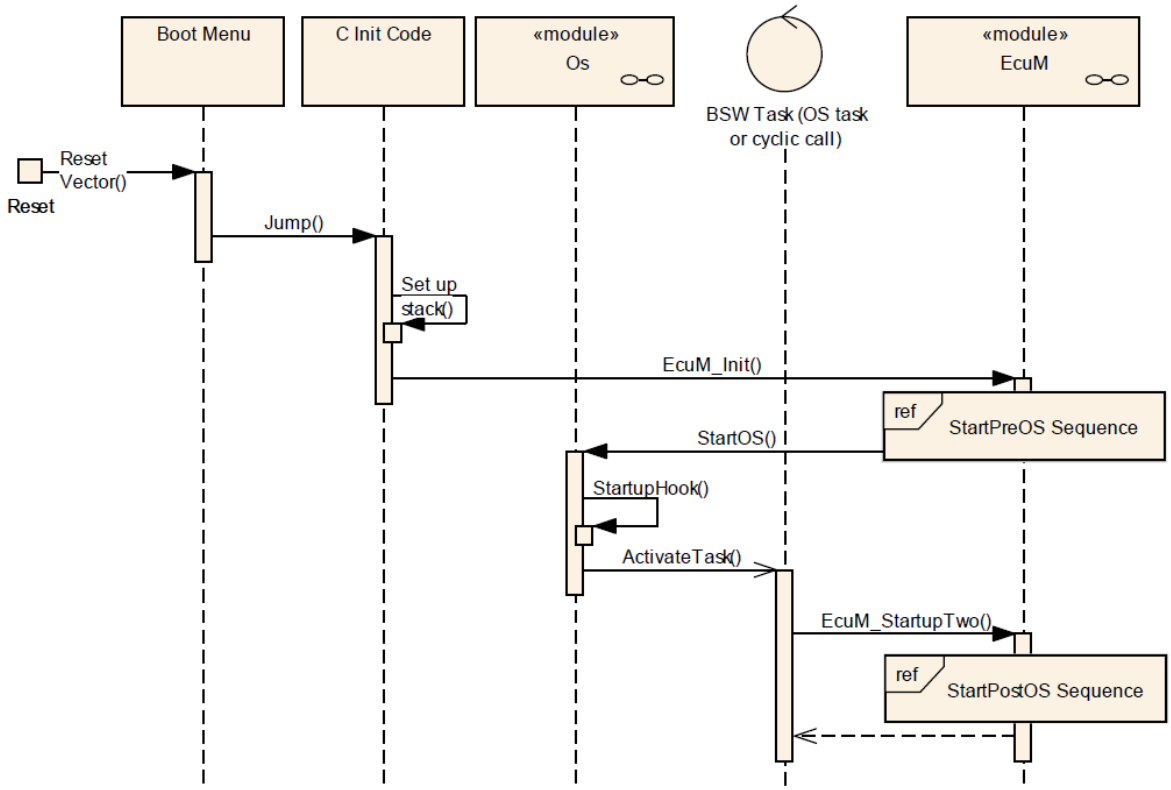

Figure 19: Startup of ECU

As a general hint it is always good to minimize the amount of code which is executed before the Os starts. Depending on the ASIL it might be required to develop all code of the startup as ASIL or to find other ways to make sure that nothing bad happened during startup e.g. by checking relevant data at a later point in time.

3.2.5.2 Shutdown

For the shutdown we have to distinguish different scenarios. From AUTOSAR perspective the EcuM also handles the shutdown. Compared with the startup we have a situation where the memory protection is enabled also during shutdown.

3.2.5.3 Sleep / Wakeup

In AUTOSAR EcuM takes also care for the sleep / wakeup handling. If a system has specific safety requirements in this area, also the EcuM shall take care. E.g. check if users are allowed to trigger a sleep / do a wakeup validation.

3.2.6 Error handling

When BSW modules are mapped to different partitions they do not change the overall AUTOSAR error handling. E.g. calls to Dem or Det still take place and – depending on the mapping – may cross partition boundaries.
Nevertheless the use of more than one partition with BSW modules introduces some new fault scenarios:

- A BSW function located in a trusted memory protected partition may cause a memory violation.
- A BSW function may be executed with timing protection and may run out of time, causing a timing violation.

- A BSW function may try to access some hardware registers where it has no access to.
- …

In AUTOSAR systems without BSW distribution these issues are normally not detected because the timing protection is not used for BSW tasks. This may cause problems during normal program execution probably or at a later point in time.
In a partitioned system where the protection is enabled also for BSW modules the problems are detected and reported via the OsProtectionHook. Although it is possible to restart a single OsApplication, restarting of single BSW partitions is not possible, since the BSW as whole has too many dependencies between the modules. This means that also for partitioned systems a protection fault is fatal and will cause a restart of the system. The advantage is that the fault can be detected much earlier and the restart can be made in a more controlled manner.

3.2.7 Timing protection

From the errors mentioned in 3.2.6 the timing faults are a special case since they may happen at any time. E.g. consider the following example:

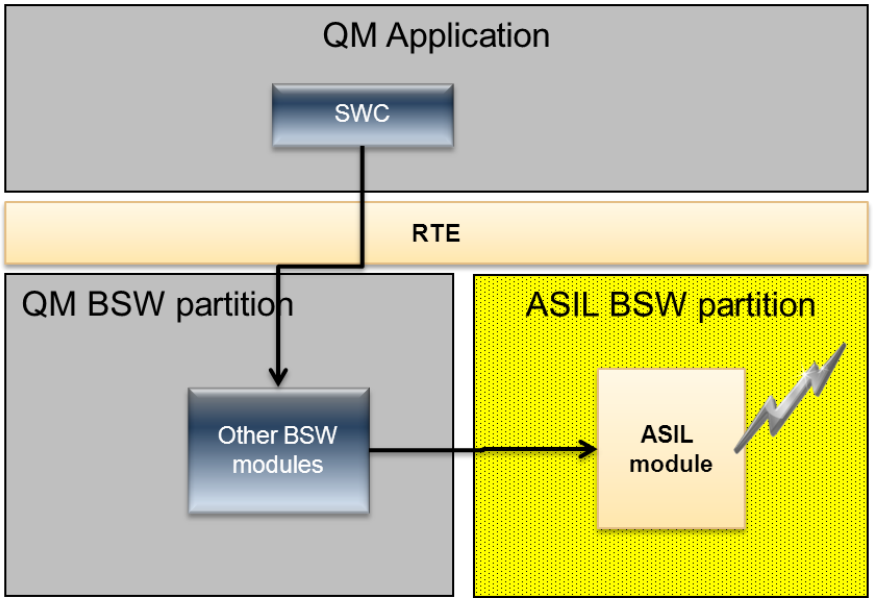

Figure 20: Timing fault

Here a runnable of a SWC calls an AUTOSAR service and continues execution in the QM BSW partition. From here a call to an ASIL module located in a different partition is performed. Then – right within the ASIL module – the timing violation takes place. The ASIL module has no chance to detect the problem and the system will shutdown.

To avoid such scenarios, trusted OsApplications have the ability to delay timing violation up to the point when the causing task (or ISR) leaves the partition. If both BSW partitions have the flag enabled the timing violation is reported at the point where the call from the SWC to the BSW module returns. Then it causes a violation and may end with a restart of the QM Application partition. The advantage here is that the BSW does not report the issue and there is no need for a shutdown.

The feature can be enabled for each trusted OsApplication via the configuration parameter `OsTrustedApplicationDelayTimingViolationCall`.

3.2.8 Combining Safety and Multi-Core

In case ASIL systems are implemented using a multi-core architecture, all considerations made until now for both, safety and multi-core, are valid. In a multi-core system, the BSW is assigned to core specific partitions. If safety is added, we have core specific QM partitions (one per core) and core specific ASIL partitions. The specific multi-core configuration parameters and the specific safety configuration parameters are independent and need to be set according to the multi-core respectively safety needs.

3.2.9 Performance Considerations

The main goal for BSW distribution within safety systems is the minimized effort if only (small) parts of the system need to be developed according to ASIL. The drawback is that the protection schema causes additional overhead. The amount of time required for the overhead depends on the project and on the mapping of the BSW modules and the frequency of interaction between the partitions.

The overhead will be minimized if …

- … as few as possible BSW partitions are used. Adding more partitions causes in all cases more overhead.
- … mapping of BSW modules follows the “nearest” approach. This means that modules with a high interaction should be placed in one partition. E.g. placing the whole communication stack in on partition is much faster than splitting it up and placing e.g. the PduR in a separate partition.
- … the number of inter partition calls is minimized. The possibilities for the user are normally limited since AUTOSAR defines the interaction between the BSW modules. Nevertheless integrator code and CDDs can be written in such way that the number of such inter partition calls is minimal.
- … specific hardware features are supported. E.g. if there is a possibility to have more memory regions by hardware they can be utilized to avoid copying data for OUT or INOUT parameters. Note that it is not enough that the hardware offers such mechanisms; the AUTOSAR vendor must also utilize it (e.g. by supporting such features in the Os or memory mapping handling).
- … avoid IOC calls. IOC will always do a copy of your data. Thus avoiding calls to it will increase the performance. In general try to “pull” the data instead of “push”, this means the caller shall (after return of CallTrustedFunction()) try to read the data. The buffer shall be on the callee side if possible.

3.2.10 Constraints

The approach to separate BSW modules into different partitions works, but has limitations depending on the available hardware:

- On some MCUs the access to registers is limited to specific processor modes. In such cases a peek/poke approach (see 3.2.4) is usable but consumes more time than a direct access. The amount of time spend for these functions may be fine for startup or shutdown, but not during normal operation if performed with high frequency.
- Normally only write access is limited between (BSW) partitions. Sometimes even a read access to peripheral registers has write effects (e.g. reading the buffer of received characters). In such cases also the read access may be limited.
- Sometimes the hardware does not support the use of memory protection while executing in privileged modes. In such cases it is recommended to run all partitions in non-privileged modes to use memory protection. The amount of code which requires privilege modes shall be minimal in such cases.

Note that for those measures typically the MCAL vendor is responsible. This may also apply for an MCAL qualified to an ASIL if the BSW is only QM.

4 Outlook on Upcoming AUTOSAR Versions

In this chapter, we list changes to the distribution of BSW that may occur in the next backward incompatible release of AUTOSAR. Hence, the content of this chapter is not applicable to AUTOSAR 4.x implementations, but is supposed to show possible extensions and enhancements for future versions of AUTOSAR. Note that all these topics need to be considered in parallel, because definitions of BSW functional clusters and their standardized interfaces, which will be named "Standardized AUTOSAR BSW Cluster Interface" then, are needed to support a safety use case.

4.1 Known limitations

The support for Basic Software Allocation in AUTOSAR is currently limited to backward compatible changes (w.r.t. AUTOSAR 4.0.3). This currently results in the following restrictions, which may not apply to future releases of AUTOSAR:

- There is only one QM BSW partition per core.
- Communication between master and satellites is not standardized.
- BSW functional clusters and their AUTOSAR BSW Cluster Interface are not standardized.

4.2 Inter BSW module calls in distributed BSW

Currently the BSW distribution has the constraint that existing QM modules shall be reused as is. If we would weaken this we can allow a more performant communication between modules. E.g. it could be possible to include SchM_Call()s directly at the caller and to avoid the stubs. (Typically the caller knows the context of the call and can prepare the best environment for the call).
Also multi-core systems would benefit if all inter BSW module calls are encapsulated with a SchM_Call().

4.3 Standardized BSW functional clusters

BSW functional clusters are groups of functionally coherent BSW modules. Each BSW functional cluster includes a set of BSW modules. It is possible to have several functional clusters of the same type (e.g. several I/O clusters in different partitions), each using a different set of modules (e.g. IOHWA + ADC in one partition and IOHWA + ADC + DIO in the second partition). Each functional cluster has a ”AUTOSAR BSW Cluster Interface", which is used to communicate with other functional clusters

BSW functional clusters can be allocated to different partitions, and functional clusters of the same type can be available in several partitions. Different functional clusters can be allocated to the same or to different partitions.

The same functional cluster can only exist at most once in each partition.

But this whole cluster allocation and the resulting real interfaces are not yet standardized, just the technique is proposed here. Thus:

Upcoming versions of AUTOSAR may standardize one or more of the following:

- Define which modules are assigned to which BSW functional cluster (=> “Standardized BSW functional cluster”). It is very likely that modules of the

same stack (for instance I/O services, I/O hardware abstraction and I/O drivers) will be assigned to the same functional cluster.

- Standardize communication between functional clusters of different types via "Standardized AUTOSAR BSW cluster interfaces", as shown in Figure 21.

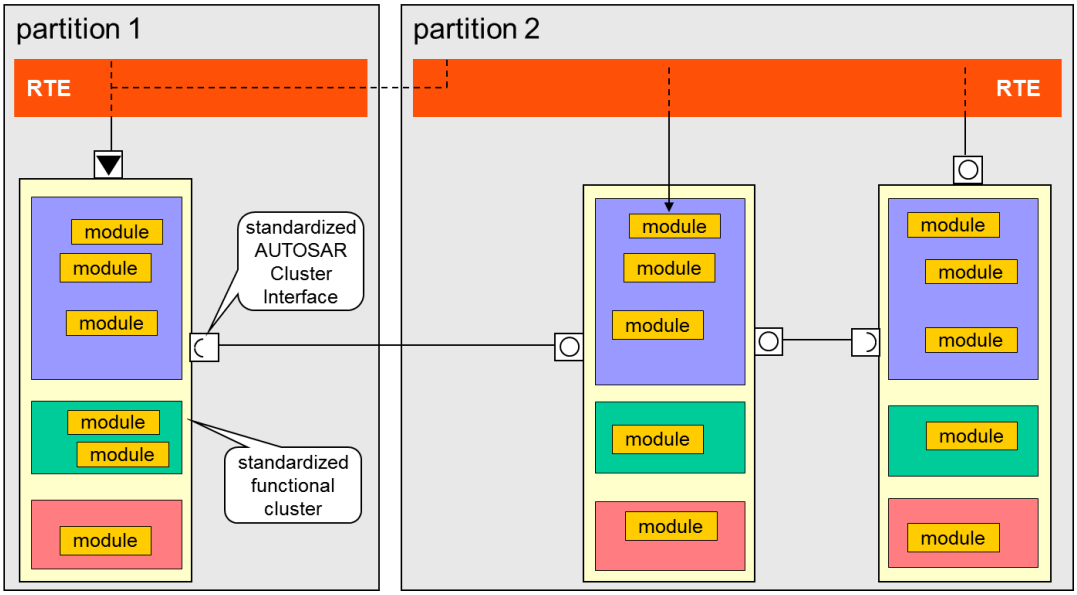

Figure 21: Standardized BSW Functional Clusters

5 Glossary

All technical terms used throughout this document - except the ones listed here - can be found in the official AUTOSAR glossary [2] or the Software Component Template Specification [3].

5.1 Acronyms and abbreviations

Abbreviation	Explanation
ASIL	Automotive Safety Integrity Level
QM	Quality Managed (i.e. not developed according to ASIL requirements)
IOC	Inter OS-Application communicator, part of OS
MCU	microcontroller unit, µC
MCAL	microcontroller abstraction layer

5.2 Technical Terms

Term	Explanation
BSW functional cluster	A coherent group of BSW modules. The technique is proposed in this document, but a real allocation of modules to clusters is currently not standardized. A BSW functional cluster may be similar to what usually is called a "stack", but it would also be possible to combine several stacks into a cluster or to distribute a stack across several clusters. A BSW functional cluster includes the superset of modules, which can be part of the functional cluster, but not all modules need to be available in a specific implementation. In case the real allocation of BSW modules to BSW functional clusters is standardized in future, they probably will be named “Standardized BSW functional clusters”. BSW functional clusters can be allocated to different partitions, and clusters of the same type can be available in several partitions (either on the same or on different cores). Different functional clusters can be allocated to the same partition. Note: Contrary to ICC2 clustering, the internal structure and the interfaces between the modules within the functional cluster are not affected by the BSW multi-core support in AUTOSAR 4.1.1.
AUTOSAR BSW Cluster Interface	Interfaces between BSW functional clusters resulting from a vendor/project specific definition of BSW functional clusters. The technique is proposed in this document in a vendor/project specific way. But the allocation of modules to BSW functional clusters and thus the resulting interfaces are not standardized yet (if possible at all). This term may be defined in an upcoming release of AUTOSAR as “Standardized AUTOSAR BSW Cluster Interface” after standardization. Contrary to the standardized AUTOSAR interfaces, AUTOSAR BSW Cluster Interfaces shall not be connected to SW-Cs or BSW modules on other MCUs.
Master	Part of a distributed BSW module that coordinates requests by satellites and can filter or monitor incoming satellite requests. The

	master may work properly even if the satellites are not available. In future versions of AUTOSAR, where case partitioning may be used to enhance safety, it may be recommended or mandatory to locate the master in a partition with a high trust level, e.g. in a trusted partition.
Satellite	Part of a distributed BSW module. The distribution of work between master and satellite is implementation specific. One possibility is that the satellite only provides the interfaces to the other modules and routes all requests to the master and answers back to the other modules. In a different scenario, the satellite can provide the full functionality locally and only synchronizes its internal states with the master if necessary. Intermediate forms between these two scenarios are possible, but the satellites in general cannot work without the master.

6 References

[1] Requirements on Basic Software Module Description Template
AUTOSAR_RS_BSWModuleDescriptionTemplate

[2] Glossary
AUTOSAR_TR_Glossary

[3] Software Component Template
AUTOSAR_TPS_SoftwareComponentTemplate

[4] Concept Enhanced BSW Allocation
AUTOSAR_CONC_EnhancedBSWAllocation

[5] Specification of Basic Software Mode Manager
AUTOSAR_SWS_BSWModeManager